

---

# **pandas: powerful Python data analysis toolkit**

***Release 0.23.2***

**Wes McKinney & PyData Development Team**

**Jul 05, 2018**



# CONTENTS

<b>1</b>	<b>What's New</b>	<b>3</b>
1.1	v0.23.2	3
1.1.1	Logical Reductions over Entire DataFrame	3
1.1.2	Fixed Regressions	4
1.1.3	Build Changes	4
1.1.4	Bug Fixes	4
1.2	v0.23.1	5
1.2.1	Fixed Regressions	5
1.2.2	Performance Improvements	6
1.2.3	Bug Fixes	6
1.3	v0.23.0 (May 15, 2018)	8
1.3.1	New features	10
1.3.1.1	JSON read/write round-trippable with <code>orient='table'</code>	10
1.3.1.2	<code>.assign()</code> accepts dependent arguments	11
1.3.1.3	Merging on a combination of columns and index levels	12
1.3.1.4	Sorting by a combination of columns and index levels	13
1.3.1.5	Extending Pandas with Custom Types (Experimental)	13
1.3.1.6	New observed keyword for excluding unobserved categories in <code>groupby</code>	14
1.3.1.7	Rolling/Expanding.apply() accepts <code>raw=False</code> to pass a <code>Series</code> to the function	16
1.3.1.8	<code>DataFrame.interpolate</code> has gained the <code>limit_area</code> kwarg	17
1.3.1.9	<code>get_dummies</code> now supports <code>dtype</code> argument	18
1.3.1.10	<code>Timedelta</code> mod method	18
1.3.1.11	<code>.rank()</code> handles <code>inf</code> values when <code>NaN</code> are present	19
1.3.1.12	<code>Series.str.cat</code> has gained the <code>join</code> kwarg	20
1.3.1.13	<code>DataFrame.astype</code> performs column-wise conversion to <code>Categorical</code>	21
1.3.1.14	Other Enhancements	21
1.3.2	Backwards incompatible API changes	23
1.3.2.1	Dependencies have increased minimum versions	23
1.3.2.2	Instantiation from dicts preserves dict insertion order for python 3.6+	24
1.3.2.3	Deprecate Panel	24
1.3.2.4	<code>pandas.core.common</code> removals	25
1.3.2.5	Changes to make output of <code>DataFrame.apply</code> consistent	26
1.3.2.6	Concatenation will no longer sort	28
1.3.2.7	Build Changes	28
1.3.2.8	Index Division By Zero Fills Correctly	28
1.3.2.9	Extraction of matching patterns from strings	29
1.3.2.10	Default value for the <code>ordered</code> parameter of <code>CategoricalDtype</code>	30
1.3.2.11	Better pretty-printing of DataFrames in a terminal	31
1.3.2.12	Datetimelike API Changes	32
1.3.2.13	Other API Changes	33

1.3.3	Deprecations	35
1.3.4	Removal of prior version deprecations/changes	36
1.3.5	Performance Improvements	37
1.3.6	Documentation Changes	38
1.3.7	Bug Fixes	39
1.3.7.1	Categorical	39
1.3.7.2	Datetimelike	39
1.3.7.3	Timedelta	40
1.3.7.4	Timezones	41
1.3.7.5	Offsets	42
1.3.7.6	Numeric	42
1.3.7.7	Strings	42
1.3.7.8	Indexing	42
1.3.7.9	MultiIndex	43
1.3.7.10	I/O	44
1.3.7.11	Plotting	45
1.3.7.12	Groupby/Resample/Rolling	45
1.3.7.13	Sparse	46
1.3.7.14	Reshaping	46
1.3.7.15	Other	47
1.4	v0.22.0 (December 29, 2017)	48
1.4.1	Backwards incompatible API changes	48
1.4.1.1	Arithmetic Operations	48
1.4.1.2	Grouping by a Categorical	49
1.4.1.3	Resample	49
1.4.1.4	Rolling and Expanding	51
1.4.2	Compatibility	51
1.5	v0.21.1 (December 12, 2017)	52
1.5.1	Restore Matplotlib datetime Converter Registration	52
1.5.2	New features	53
1.5.2.1	Improvements to the Parquet IO functionality	53
1.5.2.2	Other Enhancements	53
1.5.3	Deprecations	53
1.5.4	Performance Improvements	53
1.5.5	Bug Fixes	53
1.5.5.1	Conversion	53
1.5.5.2	Indexing	54
1.5.5.3	I/O	54
1.5.5.4	Plotting	54
1.5.5.5	Groupby/Resample/Rolling	54
1.5.5.6	Reshaping	55
1.5.5.7	Numeric	55
1.5.5.8	Categorical	55
1.5.5.9	String	55
1.6	v0.21.0 (October 27, 2017)	55
1.6.1	New features	57
1.6.1.1	Integration with Apache Parquet file format	57
1.6.1.2	infer_objects type conversion	57
1.6.1.3	Improved warnings when attempting to create columns	58
1.6.1.4	drop now also accepts index/columns keywords	58
1.6.1.5	rename, reindex now also accept axis keyword	59
1.6.1.6	CategoricalDtype for specifying categoricals	60
1.6.1.7	GroupBy objects now have a pipe method	61
1.6.1.8	Categorical.rename_categories accepts a dict-like	62



1.6.1.9	Other Enhancements . . . . .	62
1.6.2	Backwards incompatible API changes . . . . .	64
1.6.2.1	Dependencies have increased minimum versions . . . . .	64
1.6.2.2	Sum/Prod of all-NaN or empty Series/DataFrames is now consistently NaN . . . . .	64
1.6.2.3	Indexing with a list with missing labels is Deprecated . . . . .	65
1.6.2.4	NA naming Changes . . . . .	66
1.6.2.5	Iteration of Series/Index will now return Python scalars . . . . .	66
1.6.2.6	Indexing with a Boolean Index . . . . .	67
1.6.2.7	PeriodIndex resampling . . . . .	68
1.6.2.8	Improved error handling during item assignment in pd.eval . . . . .	69
1.6.2.9	Dtype Conversions . . . . .	70
1.6.2.10	MultiIndex Constructor with a Single Level . . . . .	71
1.6.2.11	UTC Localization with Series . . . . .	71
1.6.2.12	Consistency of Range Functions . . . . .	72
1.6.2.13	No Automatic Matplotlib Converters . . . . .	73
1.6.2.14	Other API Changes . . . . .	73
1.6.3	Deprecations . . . . .	74
1.6.3.1	Series.select and DataFrame.select . . . . .	75
1.6.3.2	Series.argmax and Series.argmin . . . . .	75
1.6.4	Removal of prior version deprecations/changes . . . . .	75
1.6.5	Performance Improvements . . . . .	76
1.6.6	Documentation Changes . . . . .	76
1.6.7	Bug Fixes . . . . .	76
1.6.7.1	Conversion . . . . .	76
1.6.7.2	Indexing . . . . .	77
1.6.7.3	I/O . . . . .	77
1.6.7.4	Plotting . . . . .	78
1.6.7.5	Groupby/Resample/Rolling . . . . .	78
1.6.7.6	Sparse . . . . .	79
1.6.7.7	Reshaping . . . . .	79
1.6.7.8	Numeric . . . . .	80
1.6.7.9	Categorical . . . . .	80
1.6.7.10	PyPy . . . . .	80
1.6.7.11	Other . . . . .	80
1.7	v0.20.3 (July 7, 2017) . . . . .	81
1.7.1	Bug Fixes . . . . .	81
1.7.1.1	Conversion . . . . .	81
1.7.1.2	Indexing . . . . .	81
1.7.1.3	I/O . . . . .	81
1.7.1.4	Plotting . . . . .	82
1.7.1.5	Reshaping . . . . .	82
1.7.1.6	Categorical . . . . .	82
1.8	v0.20.2 (June 4, 2017) . . . . .	82
1.8.1	Enhancements . . . . .	83
1.8.2	Performance Improvements . . . . .	83
1.8.3	Bug Fixes . . . . .	83
1.8.3.1	Conversion . . . . .	83
1.8.3.2	Indexing . . . . .	84
1.8.3.3	I/O . . . . .	84
1.8.3.4	Plotting . . . . .	84
1.8.3.5	Groupby/Resample/Rolling . . . . .	84
1.8.3.6	Sparse . . . . .	84
1.8.3.7	Reshaping . . . . .	85
1.8.3.8	Numeric . . . . .	85

	1.8.3.9	Categorical . . . . .	85
	1.8.3.10	Other . . . . .	85
1.9	v0.20.1 (May 5, 2017)	. . . . .	85
	1.9.1	New features . . . . .	87
	1.9.1.1	agg API for DataFrame/Series . . . . .	87
	1.9.1.2	dtype keyword for data IO . . . . .	89
	1.9.1.3	.to_datetime() has gained an origin parameter . . . . .	90
	1.9.1.4	Groupby Enhancements . . . . .	90
	1.9.1.5	Better support for compressed URLs in read_csv . . . . .	91
	1.9.1.6	Pickle file I/O now supports compression . . . . .	91
	1.9.1.7	UInt64 Support Improved . . . . .	92
	1.9.1.8	GroupBy on Categoricals . . . . .	93
	1.9.1.9	Table Schema Output . . . . .	94
	1.9.1.10	SciPy sparse matrix from/to SparseDataFrame . . . . .	94
	1.9.1.11	Excel output for styled DataFrames . . . . .	95
	1.9.1.12	IntervalIndex . . . . .	96
	1.9.1.13	Other Enhancements . . . . .	98
	1.9.2	Backwards incompatible API changes . . . . .	100
	1.9.2.1	Possible incompatibility for HDF5 formats created with pandas < 0.13.0 . . . . .	100
	1.9.2.2	Map on Index types now return other Index types . . . . .	101
	1.9.2.3	Accessing datetime fields of Index now return Index . . . . .	102
	1.9.2.4	pd.unique will now be consistent with extension types . . . . .	102
	1.9.2.5	S3 File Handling . . . . .	104
	1.9.2.6	Partial String Indexing Changes . . . . .	104
	1.9.2.7	Concat of different float dtypes will not automatically upcast . . . . .	105
	1.9.2.8	Pandas Google BigQuery support has moved . . . . .	105
	1.9.2.9	Memory Usage for Index is more Accurate . . . . .	105
	1.9.2.10	DataFrame.sort_index changes . . . . .	106
	1.9.2.11	Groupby Describe Formatting . . . . .	108
	1.9.2.12	Window Binary Corr/Cov operations return a MultiIndex DataFrame . . . . .	109
	1.9.2.13	HDFStore where string comparison . . . . .	110
	1.9.2.14	Index.intersection and inner join now preserve the order of the left Index . . . . .	110
	1.9.2.15	Pivot Table always returns a DataFrame . . . . .	111
	1.9.2.16	Other API Changes . . . . .	112
	1.9.3	Reorganization of the library: Privacy Changes . . . . .	113
	1.9.3.1	Modules Privacy Has Changed . . . . .	113
	1.9.3.2	pandas.errors . . . . .	114
	1.9.3.3	pandas.testing . . . . .	115
	1.9.3.4	pandas.plotting . . . . .	115
	1.9.3.5	Other Development Changes . . . . .	115
	1.9.4	Deprecations . . . . .	115
	1.9.4.1	Deprecate .ix . . . . .	115
	1.9.4.2	Deprecate Panel . . . . .	116
	1.9.4.3	Deprecate groupby.agg() with a dictionary when renaming . . . . .	117
	1.9.4.4	Deprecate .plotting . . . . .	119
	1.9.4.5	Other Deprecations . . . . .	119
	1.9.5	Removal of prior version deprecations/changes . . . . .	120
	1.9.6	Performance Improvements . . . . .	121
	1.9.7	Bug Fixes . . . . .	121
	1.9.7.1	Conversion . . . . .	121
	1.9.7.2	Indexing . . . . .	122
	1.9.7.3	I/O . . . . .	123
	1.9.7.4	Plotting . . . . .	124
	1.9.7.5	Groupby/Resample/Rolling . . . . .	125

	1.9.7.6	Sparse	125
	1.9.7.7	Reshaping	125
	1.9.7.8	Numeric	126
	1.9.7.9	Other	126
1.10	v0.19.2	(December 24, 2016)	126
	1.10.1	Enhancements	127
	1.10.2	Performance Improvements	127
	1.10.3	Bug Fixes	127
1.11	v0.19.1	(November 3, 2016)	128
	1.11.1	Performance Improvements	129
	1.11.2	Bug Fixes	129
1.12	v0.19.0	(October 2, 2016)	130
	1.12.1	New features	132
	1.12.1.1	<code>merge_asof</code> for asof-style time-series joining	132
	1.12.1.2	<code>.rolling()</code> is now time-series aware	134
	1.12.1.3	<code>read_csv</code> has improved support for duplicate column names	136
	1.12.1.4	<code>read_csv</code> supports parsing <code>Categorical</code> directly	137
	1.12.1.5	Categorical Concatenation	138
	1.12.1.6	Semi-Month Offsets	139
	1.12.1.7	New Index methods	139
	1.12.1.8	Google BigQuery Enhancements	140
	1.12.1.9	Fine-grained numpy errstate	141
	1.12.1.10	<code>get_dummies</code> now returns integer dtypes	141
	1.12.1.11	Downcast values to smallest possible dtype in <code>to_numeric</code>	141
	1.12.1.12	pandas development API	142
	1.12.1.13	Other enhancements	143
	1.12.2	API changes	145
	1.12.2.1	<code>Series.tolist()</code> will now return Python types	145
	1.12.2.2	<code>Series</code> operators for different indexes	146
	1.12.2.3	<code>Series</code> type promotion on assignment	149
	1.12.2.4	<code>.to_datetime()</code> changes	150
	1.12.2.5	Merging changes	150
	1.12.2.6	<code>.describe()</code> changes	151
	1.12.2.7	Period changes	153
	1.12.2.8	Index <code>+</code> / <code>-</code> no longer used for set operations	154
	1.12.2.9	Index <code>.difference</code> and <code>.symmetric_difference</code> changes	155
	1.12.2.10	Index <code>.unique</code> consistently returns Index	156
	1.12.2.11	MultiIndex constructors, <code>groupby</code> and <code>set_index</code> preserve categorical dtypes	156
	1.12.2.12	<code>read_csv</code> will progressively enumerate chunks	158
	1.12.2.13	Sparse Changes	158
	1.12.2.14	Indexer dtype changes	161
	1.12.2.15	Other API Changes	162
	1.12.3	Deprecations	163
	1.12.4	Removal of prior version deprecations/changes	163
	1.12.5	Performance Improvements	164
	1.12.6	Bug Fixes	165
1.13	v0.18.1	(May 3, 2016)	170
	1.13.1	New features	171
	1.13.1.1	Custom Business Hour	171
	1.13.1.2	<code>.groupby(...)</code> syntax with window and resample operations	172
	1.13.1.3	Method chaining improvements	174
	1.13.1.4	Partial string indexing on <code>DateTimeIndex</code> when part of a <code>MultiIndex</code>	176
	1.13.1.5	Assembling Datetimes	177
	1.13.1.6	Other Enhancements	178

1.13.2	Sparse changes	179
1.13.3	API changes	180
1.13.3.1	<code>.groupby(...).nth()</code> changes	180
1.13.3.2	numpy function compatibility	182
1.13.3.3	Using <code>.apply</code> on groupby resampling	182
1.13.3.4	Changes in <code>read_csv</code> exceptions	183
1.13.3.5	<code>to_datetime</code> error changes	184
1.13.3.6	Other API changes	185
1.13.3.7	Deprecations	185
1.13.4	Performance Improvements	185
1.13.5	Bug Fixes	186
1.14	v0.18.0 (March 13, 2016)	188
1.14.1	New features	190
1.14.1.1	Window functions are now methods	190
1.14.1.2	Changes to <code>rename</code>	192
1.14.1.3	Range Index	192
1.14.1.4	Changes to <code>str.extract</code>	193
1.14.1.5	Addition of <code>str.extractall</code>	194
1.14.1.6	Changes to <code>str.cat</code>	195
1.14.1.7	Datetimelike rounding	195
1.14.1.8	Formatting of Integers in <code>FloatIndex</code>	197
1.14.1.9	Changes to dtype assignment behaviors	198
1.14.1.10	<code>to_xarray</code>	199
1.14.1.11	Latex Representation	200
1.14.1.12	<code>pd.read_sas()</code> changes	200
1.14.1.13	Other enhancements	200
1.14.2	Backwards incompatible API changes	201
1.14.2.1	NaT and Timedelta operations	201
1.14.2.2	Changes to <code>msgpack</code>	202
1.14.2.3	Signature change for <code>.rank</code>	203
1.14.2.4	Bug in <code>QuarterBegin</code> with <code>n=0</code>	204
1.14.2.5	Resample API	204
1.14.2.6	Changes to <code>eval</code>	209
1.14.2.7	Other API Changes	211
1.14.2.8	Deprecations	212
1.14.2.9	Removal of deprecated float indexers	212
1.14.2.10	Removal of prior version deprecations/changes	215
1.14.3	Performance Improvements	215
1.14.4	Bug Fixes	215
1.15	v0.17.1 (November 21, 2015)	218
1.15.1	New features	219
1.15.1.1	Conditional HTML Formatting	219
1.15.2	Enhancements	219
1.15.3	API changes	221
1.15.3.1	Deprecations	221
1.15.4	Performance Improvements	221
1.15.5	Bug Fixes	222
1.16	v0.17.0 (October 9, 2015)	223
1.16.1	New features	225
1.16.1.1	Datetime with TZ	225
1.16.1.2	Releasing the GIL	227
1.16.1.3	Plot submethods	227
1.16.1.4	Additional methods for <code>dt</code> accessor	228
1.16.1.5	Period Frequency Enhancement	230

1.16.1.6	Support for SAS XPORT files . . . . .	230
1.16.1.7	Support for Math Functions in <code>.eval()</code> . . . . .	231
1.16.1.8	Changes to Excel with <code>MultiIndex</code> . . . . .	231
1.16.1.9	Google BigQuery Enhancements . . . . .	232
1.16.1.10	Display Alignment with Unicode East Asian Width . . . . .	233
1.16.1.11	Other enhancements . . . . .	233
1.16.2	Backwards incompatible API changes . . . . .	237
1.16.2.1	Changes to sorting API . . . . .	237
1.16.2.2	Changes to <code>to_datetime</code> and <code>to_timedelta</code> . . . . .	238
1.16.2.3	Changes to Index Comparisons . . . . .	239
1.16.2.4	Changes to Boolean Comparisons vs. <code>None</code> . . . . .	240
1.16.2.5	HDFStore dropna behavior . . . . .	241
1.16.2.6	Changes to <code>display.precision</code> option . . . . .	242
1.16.2.7	Changes to <code>Categorical.unique</code> . . . . .	242
1.16.2.8	Changes to <code>bool</code> passed as header in Parsers . . . . .	243
1.16.2.9	Other API Changes . . . . .	243
1.16.2.10	Deprecations . . . . .	244
1.16.2.11	Removal of prior version deprecations/changes . . . . .	245
1.16.3	Performance Improvements . . . . .	246
1.16.4	Bug Fixes . . . . .	247
1.17	v0.16.2 (June 12, 2015) . . . . .	250
1.17.1	New features . . . . .	251
1.17.1.1	Pipe . . . . .	251
1.17.1.2	Other Enhancements . . . . .	252
1.17.2	API Changes . . . . .	252
1.17.3	Performance Improvements . . . . .	252
1.17.4	Bug Fixes . . . . .	253
1.18	v0.16.1 (May 11, 2015) . . . . .	254
1.18.1	Enhancements . . . . .	255
1.18.1.1	<code>CategoricalIndex</code> . . . . .	255
1.18.1.2	<code>Sample</code> . . . . .	257
1.18.1.3	String Methods Enhancements . . . . .	258
1.18.1.4	Other Enhancements . . . . .	259
1.18.2	API changes . . . . .	261
1.18.2.1	Deprecations . . . . .	261
1.18.3	Index Representation . . . . .	261
1.18.4	Performance Improvements . . . . .	263
1.18.5	Bug Fixes . . . . .	263
1.19	v0.16.0 (March 22, 2015) . . . . .	265
1.19.1	New features . . . . .	266
1.19.1.1	<code>DataFrame</code> Assign . . . . .	266
1.19.1.2	Interaction with <code>scipy.sparse</code> . . . . .	267
1.19.1.3	String Methods Enhancements . . . . .	269
1.19.1.4	Other enhancements . . . . .	270
1.19.2	Backwards incompatible API changes . . . . .	271
1.19.2.1	Changes in <code>Timedelta</code> . . . . .	271
1.19.2.2	Indexing Changes . . . . .	272
1.19.2.3	Categorical Changes . . . . .	274
1.19.2.4	Other API Changes . . . . .	276
1.19.2.5	Deprecations . . . . .	278
1.19.2.6	Removal of prior version deprecations/changes . . . . .	278
1.19.3	Performance Improvements . . . . .	278
1.19.4	Bug Fixes . . . . .	279
1.20	v0.15.2 (December 12, 2014) . . . . .	281

1.20.1	API changes . . . . .	282
1.20.2	Enhancements . . . . .	284
1.20.3	Performance . . . . .	285
1.20.4	Bug Fixes . . . . .	286
1.21	v0.15.1 (November 9, 2014) . . . . .	287
1.21.1	API changes . . . . .	287
1.21.2	Enhancements . . . . .	291
1.21.3	Bug Fixes . . . . .	292
1.22	v0.15.0 (October 18, 2014) . . . . .	293
1.22.1	New features . . . . .	294
1.22.1.1	Categoricals in Series/DataFrame . . . . .	294
1.22.1.2	TimedeltaIndex/Scalar . . . . .	296
1.22.1.3	Memory Usage . . . . .	298
1.22.1.4	.dt accessor . . . . .	299
1.22.1.5	Timezone handling improvements . . . . .	302
1.22.1.6	Rolling/Expanding Moments improvements . . . . .	303
1.22.1.7	Improvements in the sql io module . . . . .	307
1.22.2	Backwards incompatible API changes . . . . .	307
1.22.2.1	Breaking changes . . . . .	307
1.22.2.2	Internal Refactoring . . . . .	313
1.22.2.3	Deprecations . . . . .	313
1.22.2.4	Removal of prior version deprecations/changes . . . . .	313
1.22.3	Enhancements . . . . .	314
1.22.4	Performance . . . . .	318
1.22.5	Bug Fixes . . . . .	318
1.23	v0.14.1 (July 11, 2014) . . . . .	322
1.23.1	API changes . . . . .	322
1.23.2	Enhancements . . . . .	323
1.23.3	Performance . . . . .	324
1.23.4	Experimental . . . . .	324
1.23.5	Bug Fixes . . . . .	325
1.24	v0.14.0 (May 31 , 2014) . . . . .	327
1.24.1	API changes . . . . .	328
1.24.2	Display Changes . . . . .	332
1.24.3	Text Parsing API Changes . . . . .	334
1.24.4	Groupby API Changes . . . . .	334
1.24.5	SQL . . . . .	337
1.24.6	MultiIndexing Using Slicers . . . . .	338
1.24.7	Plotting . . . . .	343
1.24.8	Prior Version Deprecations/Changes . . . . .	344
1.24.9	Deprecations . . . . .	344
1.24.10	Known Issues . . . . .	345
1.24.11	Enhancements . . . . .	345
1.24.12	Performance . . . . .	350
1.24.13	Experimental . . . . .	350
1.24.14	Bug Fixes . . . . .	350
1.25	v0.13.1 (February 3, 2014) . . . . .	355
1.25.1	Output Formatting Enhancements . . . . .	356
1.25.2	API changes . . . . .	357
1.25.3	Prior Version Deprecations/Changes . . . . .	359
1.25.4	Deprecations . . . . .	359
1.25.5	Enhancements . . . . .	359
1.25.6	Performance . . . . .	362
1.25.7	Experimental . . . . .	363

	1.25.8 Bug Fixes . . . . .	363
1.26	v0.13.0 (January 3, 2014) . . . . .	363
	1.26.1 API changes . . . . .	363
	1.26.2 Prior Version Deprecations/Changes . . . . .	366
	1.26.3 Deprecations . . . . .	366
	1.26.4 Indexing API Changes . . . . .	366
	1.26.5 Float64Index API Change . . . . .	368
	1.26.6 HDFStore API Changes . . . . .	370
	1.26.7 DataFrame repr Changes . . . . .	372
	1.26.8 Enhancements . . . . .	373
	1.26.9 Experimental . . . . .	380
	1.26.10 Internal Refactoring . . . . .	384
	1.26.11 Bug Fixes . . . . .	386
1.27	v0.12.0 (July 24, 2013) . . . . .	386
	1.27.1 API changes . . . . .	386
	1.27.2 I/O Enhancements . . . . .	390
	1.27.3 Other Enhancements . . . . .	392
	1.27.4 Experimental Features . . . . .	394
	1.27.5 Bug Fixes . . . . .	395
1.28	v0.11.0 (April 22, 2013) . . . . .	397
	1.28.1 Selection Choices . . . . .	398
	1.28.2 Selection Deprecations . . . . .	398
	1.28.3 Dtypes . . . . .	399
	1.28.4 Dtype Conversion . . . . .	400
	1.28.5 Dtype Gotchas . . . . .	401
	1.28.6 Datetimes Conversion . . . . .	403
	1.28.7 API changes . . . . .	405
	1.28.8 Enhancements . . . . .	405
1.29	v0.10.1 (January 22, 2013) . . . . .	407
	1.29.1 API changes . . . . .	407
	1.29.2 New features . . . . .	408
	1.29.3 HDFStore . . . . .	408
1.30	v0.10.0 (December 17, 2012) . . . . .	413
	1.30.1 File parsing new features . . . . .	413
	1.30.2 API changes . . . . .	414
	1.30.3 New features . . . . .	419
	1.30.4 Wide DataFrame Printing . . . . .	419
	1.30.5 Updated PyTables Support . . . . .	421
	1.30.6 N Dimensional Panels (Experimental) . . . . .	424
1.31	v0.9.1 (November 14, 2012) . . . . .	425
	1.31.1 New features . . . . .	425
	1.31.2 API changes . . . . .	428
1.32	v0.9.0 (October 7, 2012) . . . . .	429
	1.32.1 New features . . . . .	429
	1.32.2 API changes . . . . .	430
1.33	v0.8.1 (July 22, 2012) . . . . .	431
	1.33.1 New features . . . . .	431
	1.33.2 Performance improvements . . . . .	431
1.34	v0.8.0 (June 29, 2012) . . . . .	431
	1.34.1 Support for non-unique indexes . . . . .	432
	1.34.2 NumPy datetime64 dtype and 1.6 dependency . . . . .	432
	1.34.3 Time series changes and improvements . . . . .	432
	1.34.4 Other new features . . . . .	433
	1.34.5 New plotting methods . . . . .	434

1.34.6	Other API changes	434
1.34.7	Potential porting issues for pandas <= 0.7.3 users	434
1.35	v.0.7.3 (April 12, 2012)	436
1.35.1	New features	437
1.35.2	NA Boolean Comparison API Change	438
1.35.3	Other API Changes	439
1.36	v.0.7.2 (March 16, 2012)	440
1.36.1	New features	440
1.36.2	Performance improvements	440
1.37	v.0.7.1 (February 29, 2012)	440
1.37.1	New features	441
1.37.2	Performance improvements	441
1.38	v.0.7.0 (February 9, 2012)	441
1.38.1	New features	441
1.38.2	API Changes to integer indexing	442
1.38.3	API tweaks regarding label-based slicing	444
1.38.4	Changes to Series [ ] operator	445
1.38.5	Other API Changes	446
1.38.6	Performance improvements	446
1.39	v.0.6.1 (December 13, 2011)	447
1.39.1	New features	447
1.39.2	Performance improvements	447
1.40	v.0.6.0 (November 25, 2011)	447
1.40.1	New Features	447
1.40.2	Performance Enhancements	449
1.41	v.0.5.0 (October 24, 2011)	449
1.41.1	New Features	449
1.41.2	Performance Enhancements	450
1.42	v.0.4.3 through v0.4.1 (September 25 - October 9, 2011)	450
1.42.1	New Features	450
1.42.2	Performance Enhancements	451
<b>2</b>	<b>Installation</b>	<b>453</b>
2.1	Plan for dropping Python 2.7	453
2.2	Python version support	453
2.3	Installing pandas	453
2.3.1	Installing with Anaconda	453
2.3.2	Installing with Miniconda	454
2.3.3	Installing from PyPI	455
2.3.4	Installing with ActivePython	455
2.3.5	Installing using your Linux distribution's package manager.	455
2.3.6	Installing from source	455
2.4	Running the test suite	455
2.5	Dependencies	456
2.5.1	Recommended Dependencies	456
2.5.2	Optional Dependencies	456
<b>3</b>	<b>Contributing to pandas</b>	<b>459</b>
3.1	Where to start?	460
3.2	Bug reports and enhancement requests	460
3.3	Working with the code	461
3.3.1	Version control, Git, and GitHub	461
3.3.2	Getting started with Git	461
3.3.3	Forking	461



3.3.4	Creating a development environment . . . . .	461
3.3.4.1	Installing a C Compiler . . . . .	462
3.3.4.2	Creating a Python Environment . . . . .	462
3.3.4.3	Creating a Python Environment (pip) . . . . .	463
3.3.5	Creating a branch . . . . .	463
3.4	Contributing to the documentation . . . . .	464
3.4.1	About the <i>pandas</i> documentation . . . . .	464
3.4.1.1	<i>pandas</i> docstring guide . . . . .	464
3.4.2	How to build the <i>pandas</i> documentation . . . . .	480
3.4.2.1	Requirements . . . . .	480
3.4.2.2	Building the documentation . . . . .	480
3.4.2.3	Building master branch documentation . . . . .	481
3.5	Contributing to the code base . . . . .	481
3.5.1	Code standards . . . . .	482
3.5.1.1	C (cpplint) . . . . .	482
3.5.1.2	Python (PEP8) . . . . .	483
3.5.1.3	Backwards Compatibility . . . . .	483
3.5.2	Testing With Continuous Integration . . . . .	484
3.5.3	Test-driven development/code writing . . . . .	484
3.5.3.1	Writing tests . . . . .	485
3.5.3.2	Transitioning to <i>pytest</i> . . . . .	485
3.5.3.3	Using <i>pytest</i> . . . . .	486
3.5.4	Running the test suite . . . . .	487
3.5.5	Running the performance test suite . . . . .	488
3.5.6	Documenting your code . . . . .	489
3.6	Contributing your changes to <i>pandas</i> . . . . .	489
3.6.1	Committing your code . . . . .	489
3.6.2	Pushing your changes . . . . .	490
3.6.3	Review your code . . . . .	490
3.6.4	Finally, make the pull request . . . . .	491
3.6.5	Updating your pull request . . . . .	491
3.6.6	Delete your merged branch (optional) . . . . .	491
<b>4</b>	<b>Package overview</b> . . . . .	<b>493</b>
4.1	Data Structures . . . . .	493
4.1.1	Why more than one data structure? . . . . .	493
4.2	Mutability and copying of data . . . . .	494
4.3	Getting Support . . . . .	494
4.4	Community . . . . .	494
4.5	Project Governance . . . . .	494
4.6	Development Team . . . . .	494
4.7	Institutional Partners . . . . .	495
4.8	License . . . . .	495
<b>5</b>	<b>10 Minutes to <i>pandas</i></b> . . . . .	<b>497</b>
5.1	Object Creation . . . . .	497
5.2	Viewing Data . . . . .	499
5.3	Selection . . . . .	500
5.3.1	Getting . . . . .	500
5.3.2	Selection by Label . . . . .	501
5.3.3	Selection by Position . . . . .	502
5.3.4	Boolean Indexing . . . . .	503
5.3.5	Setting . . . . .	504
5.4	Missing Data . . . . .	505

5.5	Operations . . . . .	506
5.5.1	Stats . . . . .	506
5.5.2	Apply . . . . .	507
5.5.3	Histogramming . . . . .	507
5.5.4	String Methods . . . . .	508
5.6	Merge . . . . .	508
5.6.1	Concat . . . . .	508
5.6.2	Join . . . . .	509
5.6.3	Append . . . . .	510
5.7	Grouping . . . . .	511
5.8	Reshaping . . . . .	511
5.8.1	Stack . . . . .	512
5.8.2	Pivot Tables . . . . .	513
5.9	Time Series . . . . .	514
5.10	Categoricals . . . . .	516
5.11	Plotting . . . . .	517
5.12	Getting Data In/Out . . . . .	518
5.12.1	CSV . . . . .	518
5.12.2	HDF5 . . . . .	519
5.12.3	Excel . . . . .	519
5.13	Gotchas . . . . .	520
<b>6</b>	<b>Tutorials</b>	<b>521</b>
6.1	Internal Guides . . . . .	521
6.2	pandas Cookbook . . . . .	521
6.3	Lessons for new pandas users . . . . .	522
6.4	Practical data analysis with Python . . . . .	522
6.5	Exercises for new users . . . . .	522
6.6	Modern pandas . . . . .	523
6.7	Excel charts with pandas, vincent and xlswriter . . . . .	523
6.8	Video Tutorials . . . . .	523
6.9	Various Tutorials . . . . .	523
<b>7</b>	<b>Cookbook</b>	<b>525</b>
7.1	Idioms . . . . .	525
7.1.1	if-then... . . . .	525
7.1.2	Splitting . . . . .	526
7.1.3	Building Criteria . . . . .	527
7.2	Selection . . . . .	529
7.2.1	DataFrames . . . . .	529
7.2.2	Panels . . . . .	531
7.2.3	New Columns . . . . .	531
7.3	MultiIndexing . . . . .	532
7.3.1	Arithmetic . . . . .	533
7.3.2	Slicing . . . . .	534
7.3.3	Sorting . . . . .	536
7.3.4	Levels . . . . .	536
7.4	Missing Data . . . . .	536
7.4.1	Replace . . . . .	537
7.5	Grouping . . . . .	537
7.5.1	Expanding Data . . . . .	542
7.5.2	Splitting . . . . .	542
7.5.3	Pivot . . . . .	543
7.5.4	Apply . . . . .	545

7.6	Timeseries . . . . .	547
7.6.1	Resampling . . . . .	547
7.7	Merge . . . . .	548
7.8	Plotting . . . . .	549
7.9	Data In/Out . . . . .	550
7.9.1	CSV . . . . .	550
7.9.1.1	Reading multiple files to create a single DataFrame . . . . .	551
7.9.1.2	Parsing date components in multi-columns . . . . .	551
7.9.1.3	Skip row between header and data . . . . .	552
7.9.2	SQL . . . . .	553
7.9.3	Excel . . . . .	553
7.9.4	HTML . . . . .	553
7.9.5	HDFStore . . . . .	553
7.9.6	Binary Files . . . . .	554
7.10	Computation . . . . .	555
7.11	Timedeltas . . . . .	555
7.12	Aliasing Axis Names . . . . .	557
7.13	Creating Example Data . . . . .	558
<b>8</b>	<b>Intro to Data Structures</b>	<b>559</b>
8.1	Series . . . . .	559
8.1.1	Series is ndarray-like . . . . .	561
8.1.2	Series is dict-like . . . . .	562
8.1.3	Vectorized operations and label alignment with Series . . . . .	563
8.1.4	Name attribute . . . . .	564
8.2	DataFrame . . . . .	564
8.2.1	From dict of Series or dicts . . . . .	565
8.2.2	From dict of ndarrays / lists . . . . .	566
8.2.3	From structured or record array . . . . .	566
8.2.4	From a list of dicts . . . . .	567
8.2.5	From a dict of tuples . . . . .	567
8.2.6	From a Series . . . . .	567
8.2.7	Alternate Constructors . . . . .	567
8.2.8	Column selection, addition, deletion . . . . .	568
8.2.9	Assigning New Columns in Method Chains . . . . .	570
8.2.10	Indexing / Selection . . . . .	572
8.2.11	Data alignment and arithmetic . . . . .	573
8.2.12	Transposing . . . . .	576
8.2.13	DataFrame interoperability with NumPy functions . . . . .	576
8.2.14	Console display . . . . .	577
8.2.15	DataFrame column attribute access and IPython completion . . . . .	580
8.3	Panel . . . . .	580
8.3.1	From 3D ndarray with optional axis labels . . . . .	581
8.3.2	From dict of DataFrame objects . . . . .	581
8.3.3	From DataFrame using to_panel method . . . . .	582
8.3.4	Item selection / addition / deletion . . . . .	583
8.3.5	Transposing . . . . .	583
8.3.6	Indexing / Selection . . . . .	583
8.3.7	Squeezing . . . . .	584
8.3.8	Conversion to DataFrame . . . . .	584
8.4	Deprecate Panel . . . . .	585
<b>9</b>	<b>Essential Basic Functionality</b>	<b>589</b>
9.1	Head and Tail . . . . .	589

9.2	Attributes and the raw ndarray(s)	590
9.3	Accelerated operations	591
9.4	Flexible binary operations	591
9.4.1	Matching / broadcasting behavior	592
9.4.2	Missing data / operations with fill values	595
9.4.3	Flexible Comparisons	596
9.4.4	Boolean Reductions	596
9.4.5	Comparing if objects are equivalent	598
9.4.6	Comparing array-like objects	598
9.4.7	Combining overlapping data sets	599
9.4.8	General DataFrame Combine	600
9.5	Descriptive statistics	601
9.5.1	Summarizing data: describe	603
9.5.2	Index of Min/Max Values	606
9.5.3	Value counts (histogramming) / Mode	607
9.5.4	Discretization and quantiling	608
9.6	Function application	609
9.6.1	Tablewise Function Application	609
9.6.2	Row or Column-wise Function Application	611
9.6.3	Aggregation API	613
9.6.3.1	Aggregating with multiple functions	614
9.6.3.2	Aggregating with a dict	615
9.6.3.3	Mixed Dtypes	615
9.6.3.4	Custom describe	616
9.6.4	Transform API	616
9.6.4.1	Transform with multiple functions	618
9.6.4.2	Transforming with a dict	619
9.6.5	Applying Elementwise Functions	619
9.6.6	Applying with a Panel	621
9.7	Reindexing and altering labels	623
9.7.1	Reindexing to align with another object	625
9.7.2	Aligning objects with each other with <code>align</code>	626
9.7.3	Filling while reindexing	628
9.7.4	Limits on filling while reindexing	629
9.7.5	Dropping labels from an axis	630
9.7.6	Renaming / mapping labels	631
9.8	Iteration	632
9.8.1	<code>iteritems</code>	633
9.8.2	<code>iterrows</code>	634
9.8.3	<code>itertuples</code>	635
9.9	<code>.dt</code> accessor	636
9.10	Vectorized string methods	639
9.11	Sorting	640
9.11.1	By Index	640
9.11.2	By Values	641
9.11.3	By Indexes and Values	642
9.11.4	<code>searchsorted</code>	643
9.11.5	<code>smallest / largest values</code>	643
9.11.6	Sorting by a multi-index column	645
9.12	Copying	645
9.13	<code>dtypes</code>	645
9.13.1	<code>defaults</code>	648
9.13.2	<code>upcasting</code>	648
9.13.3	<code>astype</code>	649

9.13.4	object conversion . . . . .	651
9.13.5	gotchas . . . . .	654
9.14	Selecting columns based on dtype . . . . .	655
<b>10</b>	<b>Working with Text Data</b>	<b>659</b>
10.1	Splitting and Replacing Strings . . . . .	661
10.2	Concatenation . . . . .	664
10.2.1	Concatenating a single Series into a string . . . . .	664
10.2.2	Concatenating a Series and something list-like into a Series . . . . .	665
10.2.3	Concatenating a Series and something array-like into a Series . . . . .	665
10.2.4	Concatenating a Series and an indexed object into a Series, with alignment . . . . .	666
10.2.5	Concatenating a Series and many objects into a Series . . . . .	668
10.3	Indexing with .str . . . . .	670
10.4	Extracting Substrings . . . . .	670
10.4.1	Extract first match in each subject (extract) . . . . .	670
10.4.2	Extract all matches in each subject (extractall) . . . . .	672
10.5	Testing for Strings that Match or Contain a Pattern . . . . .	674
10.6	Creating Indicator Variables . . . . .	675
10.7	Method Summary . . . . .	675
<b>11</b>	<b>Options and Settings</b>	<b>679</b>
11.1	Overview . . . . .	679
11.2	Getting and Setting Options . . . . .	680
11.3	Setting Startup Options in python/ipython Environment . . . . .	681
11.4	Frequently Used Options . . . . .	681
11.5	Available Options . . . . .	687
11.6	Number Formatting . . . . .	688
11.7	Unicode Formatting . . . . .	689
11.8	Table Schema Display . . . . .	690
<b>12</b>	<b>Indexing and Selecting Data</b>	<b>691</b>
12.1	Different Choices for Indexing . . . . .	691
12.2	Basics . . . . .	692
12.3	Attribute Access . . . . .	695
12.4	Slicing ranges . . . . .	697
12.5	Selection By Label . . . . .	698
12.5.1	Slicing with labels . . . . .	701
12.6	Selection By Position . . . . .	702
12.7	Selection By Callable . . . . .	706
12.8	IX Indexer is Deprecated . . . . .	708
12.9	Indexing with list with missing labels is Deprecated . . . . .	709
12.9.1	Reindexing . . . . .	710
12.10	Selecting Random Samples . . . . .	711
12.11	Setting With Enlargement . . . . .	713
12.12	Fast scalar value getting and setting . . . . .	714
12.13	Boolean indexing . . . . .	715
12.14	Indexing with isin . . . . .	716
12.15	The where() Method and Masking . . . . .	719
12.15.1	Mask . . . . .	722
12.16	The query() Method . . . . .	723
12.16.1	MultiIndex query() Syntax . . . . .	725
12.16.2	query() Use Cases . . . . .	726
12.16.3	query() Python versus pandas Syntax Comparison . . . . .	727
12.16.4	The in and not in operators . . . . .	728

12.16.5	Special use of the <code>==</code> operator with <code>list</code> objects	730
12.16.6	Boolean Operators	732
12.16.7	Performance of <code>query()</code>	733
12.17	Duplicate Data	733
12.18	Dictionary-like <code>get()</code> method	736
12.19	The <code>lookup()</code> Method	737
12.20	Index objects	737
12.20.1	Setting metadata	738
12.20.2	Set operations on Index objects	739
12.20.3	Missing values	739
12.21	Set / Reset Index	740
12.21.1	Set an index	740
12.21.2	Reset the index	741
12.21.3	Adding an ad hoc index	742
12.22	Returning a view versus a copy	742
12.22.1	Why does assignment fail when using chained indexing?	743
12.22.2	Evaluation order matters	744
<b>13</b>	<b>MultiIndex / Advanced Indexing</b>	<b>747</b>
13.1	Hierarchical indexing (MultiIndex)	747
13.1.1	Creating a MultiIndex (hierarchical index) object	747
13.1.2	Reconstructing the level labels	750
13.1.3	Basic indexing on axis with MultiIndex	750
13.1.4	Defined Levels	751
13.1.5	Data alignment and using <code>reindex</code>	752
13.2	Advanced indexing with hierarchical index	753
13.2.1	Using slicers	755
13.2.2	Cross-section	759
13.2.3	Advanced reindexing and alignment	761
13.2.4	Swapping levels with <code>swaplevel()</code>	762
13.2.5	Reordering levels with <code>reorder_levels()</code>	762
13.3	Sorting a MultiIndex	763
13.4	Take Methods	766
13.5	Index Types	767
13.5.1	CategoricalIndex	767
13.5.2	Int64Index and RangeIndex	770
13.5.3	Float64Index	770
13.5.4	IntervalIndex	773
13.5.4.1	Generating Ranges of Intervals	774
13.6	Miscellaneous indexing FAQ	776
13.6.1	Integer indexing	776
13.6.2	Non-monotonic indexes require exact matches	776
13.6.3	Endpoints are inclusive	778
13.6.4	Indexing potentially changes underlying Series dtype	778
<b>14</b>	<b>Computational tools</b>	<b>781</b>
14.1	Statistical Functions	781
14.1.1	Percent Change	781
14.1.2	Covariance	781
14.1.3	Correlation	783
14.1.4	Data ranking	784
14.2	Window Functions	785
14.2.1	Method Summary	788
14.2.2	Rolling Windows	790

14.2.3	Time-aware Rolling . . . . .	792
14.2.4	Rolling Window Endpoints . . . . .	794
14.2.5	Time-aware Rolling vs. Resampling . . . . .	795
14.2.6	Centering Windows . . . . .	795
14.2.7	Binary Window Functions . . . . .	796
14.2.8	Computing rolling pairwise covariances and correlations . . . . .	797
14.3	Aggregation . . . . .	799
14.3.1	Applying multiple functions . . . . .	801
14.3.2	Applying different functions to DataFrame columns . . . . .	802
14.4	Expanding Windows . . . . .	803
14.4.1	Method Summary . . . . .	804
14.5	Exponentially Weighted Windows . . . . .	806
<b>15</b>	<b>Working with missing data</b>	<b>811</b>
15.1	Missing data basics . . . . .	811
15.1.1	When / why does data become missing? . . . . .	811
15.1.2	Values considered “missing” . . . . .	812
15.2	Datetimes . . . . .	813
15.3	Inserting missing data . . . . .	814
15.4	Calculations with missing data . . . . .	815
15.4.1	Sum/Prod of Empties/Nans . . . . .	816
15.4.2	NA values in GroupBy . . . . .	817
15.5	Cleaning / filling missing data . . . . .	817
15.5.1	Filling missing values: fillna . . . . .	817
15.5.2	Filling with a PandasObject . . . . .	819
15.5.3	Dropping axis labels with missing data: dropna . . . . .	820
15.5.4	Interpolation . . . . .	821
15.5.4.1	Interpolation Limits . . . . .	827
15.5.5	Replacing Generic Values . . . . .	829
15.5.6	String/Regular Expression Replacement . . . . .	830
15.5.7	Numeric Replacement . . . . .	832
15.6	Missing data casting rules and indexing . . . . .	834
<b>16</b>	<b>Group By: split-apply-combine</b>	<b>837</b>
16.1	Splitting an object into groups . . . . .	838
16.1.1	GroupBy sorting . . . . .	840
16.1.2	GroupBy object attributes . . . . .	840
16.1.3	GroupBy with MultiIndex . . . . .	841
16.1.4	Grouping DataFrame with Index Levels and Columns . . . . .	843
16.1.5	DataFrame column selection in GroupBy . . . . .	844
16.2	Iterating through groups . . . . .	845
16.3	Selecting a group . . . . .	846
16.4	Aggregation . . . . .	846
16.4.1	Applying multiple functions at once . . . . .	848
16.4.2	Applying different functions to DataFrame columns . . . . .	849
16.4.3	Cython-optimized aggregation functions . . . . .	850
16.5	Transformation . . . . .	851
16.5.1	New syntax to window and resample operations . . . . .	856
16.6	Filtration . . . . .	858
16.7	Dispatching to instance methods . . . . .	860
16.8	Flexible apply . . . . .	861
16.9	Other useful features . . . . .	864
16.9.1	Automatic exclusion of “nuisance” columns . . . . .	864
16.9.2	Handling of (un)observed Categorical values . . . . .	864

16.9.3	NA and NaT group handling	865
16.9.4	Grouping with ordered factors	865
16.9.5	Grouping with a Grouper specification	865
16.9.6	Taking the first rows of each group	867
16.9.7	Taking the nth row of each group	867
16.9.8	Enumerate group items	869
16.9.9	Enumerate groups	870
16.9.10	Plotting	871
16.9.11	Piping function calls	872
16.10	Examples	873
16.10.1	Regrouping by factor	873
16.10.2	Multi-column factorization	873
16.10.3	Groupby by Indexer to ‘resample’ data	874
16.10.4	Returning a Series to propagate names	874
<b>17</b>	<b>Merge, join, and concatenate</b>	<b>877</b>
17.1	Concatenating objects	877
17.1.1	Set logic on the other axes	880
17.1.2	Concatenating using <code>append</code>	881
17.1.3	Ignoring indexes on the concatenation axis	883
17.1.4	Concatenating with mixed ndims	883
17.1.5	More concatenating with group keys	885
17.1.6	Appending rows to a DataFrame	888
17.2	Database-style DataFrame joining/merging	889
17.2.1	Brief primer on merge methods (relational algebra)	891
17.2.2	Checking for duplicate keys	894
17.2.3	The merge indicator	894
17.2.4	Merge Dtypes	895
17.2.5	Joining on index	898
17.2.6	Joining key columns on an index	899
17.2.7	Joining a single Index to a Multi-index	901
17.2.8	Joining with two multi-indexes	902
17.2.9	Merging on a combination of columns and index levels	902
17.2.10	Overlapping value columns	903
17.2.11	Joining multiple DataFrame or Panel objects	904
17.2.12	Merging together values within Series or DataFrame columns	905
17.3	Timeseries friendly merging	905
17.3.1	Merging Ordered Data	905
17.3.2	Merging AsOf	906
<b>18</b>	<b>Reshaping and Pivot Tables</b>	<b>909</b>
18.1	Reshaping by pivoting DataFrame objects	909
18.2	Reshaping by stacking and unstacking	911
18.2.1	Multiple Levels	915
18.2.2	Missing Data	916
18.2.3	With a MultiIndex	918
18.3	Reshaping by Melt	919
18.4	Combining with stats and GroupBy	921
18.5	Pivot tables	922
18.5.1	Adding margins	925
18.6	Cross tabulations	925
18.6.1	Normalization	926
18.6.2	Adding Margins	927
18.7	Tiling	927



18.8	Computing indicator / dummy variables . . . . .	928
18.9	Factorizing values . . . . .	931
<b>19</b>	<b>Time Series / Date functionality</b>	<b>933</b>
19.1	Overview . . . . .	934
19.2	Timestamps vs. Time Spans . . . . .	934
19.3	Converting to Timestamps . . . . .	936
19.3.1	Providing a Format Argument . . . . .	936
19.3.2	Assembling Datetime from Multiple DataFrame Columns . . . . .	937
19.3.3	Invalid Data . . . . .	937
19.3.4	Epoch Timestamps . . . . .	937
19.3.5	From Timestamps to Epoch . . . . .	938
19.3.6	Using the <code>origin</code> Parameter . . . . .	939
19.4	Generating Ranges of Timestamps . . . . .	939
19.4.1	Custom Frequency Ranges . . . . .	942
19.5	Timestamp Limitations . . . . .	942
19.6	Indexing . . . . .	943
19.6.1	Partial String Indexing . . . . .	944
19.6.2	Slice vs. Exact Match . . . . .	948
19.6.3	Exact Indexing . . . . .	950
19.6.4	Truncating & Fancy Indexing . . . . .	951
19.7	Time/Date Components . . . . .	951
19.8	DateOffset Objects . . . . .	952
19.8.1	Parametric Offsets . . . . .	954
19.8.2	Using Offsets with <code>Series / DatetimeIndex</code> . . . . .	955
19.8.3	Custom Business Days . . . . .	956
19.8.4	Business Hour . . . . .	958
19.8.5	Custom Business Hour . . . . .	960
19.8.6	Offset Aliases . . . . .	961
19.8.7	Combining Aliases . . . . .	961
19.8.8	Anchored Offsets . . . . .	962
19.8.9	Anchored Offset Semantics . . . . .	963
19.8.10	Holidays / Holiday Calendars . . . . .	965
19.9	Time Series-Related Instance Methods . . . . .	967
19.9.1	Shifting / Lagging . . . . .	967
19.9.2	Frequency Conversion . . . . .	968
19.9.3	Filling Forward / Backward . . . . .	968
19.9.4	Converting to Python Datetimes . . . . .	969
19.10	Resampling . . . . .	969
19.10.1	Basics . . . . .	969
19.10.2	Upsampling . . . . .	971
19.10.3	Sparse Resampling . . . . .	972
19.10.4	Aggregation . . . . .	973
19.11	Time Span Representation . . . . .	976
19.11.1	Period . . . . .	976
19.11.2	PeriodIndex and <code>period_range</code> . . . . .	977
19.11.3	Period Dtypes . . . . .	979
19.11.4	PeriodIndex Partial String Indexing . . . . .	980
19.11.5	Frequency Conversion and Resampling with <code>PeriodIndex</code> . . . . .	982
19.12	Converting Between Representations . . . . .	983
19.13	Representing Out-of-Bounds Spans . . . . .	984
19.14	Time Zone Handling . . . . .	985
19.14.1	Working with Time Zones . . . . .	985
19.14.2	Ambiguous Times when Localizing . . . . .	990

19.14.3	TZ Aware Dtypes	992
<b>20</b>	<b>Time Deltas</b>	<b>995</b>
20.1	Parsing	995
20.1.1	to_timedelta	996
20.1.2	Timedelta limitations	997
20.2	Operations	997
20.3	Reductions	1001
20.4	Frequency Conversion	1002
20.5	Attributes	1004
20.6	TimedeltaIndex	1005
20.6.1	Generating Ranges of Time Deltas	1006
20.6.2	Using the TimedeltaIndex	1007
20.6.3	Operations	1008
20.6.4	Conversions	1009
20.7	Resampling	1009
<b>21</b>	<b>Categorical Data</b>	<b>1011</b>
21.1	Object Creation	1011
21.1.1	Series Creation	1011
21.1.2	DataFrame Creation	1013
21.1.3	Controlling Behavior	1014
21.1.4	Regaining Original Data	1016
21.2	CategoricalDtype	1016
21.2.1	Equality Semantics	1017
21.3	Description	1018
21.4	Working with categories	1018
21.4.1	Renaming categories	1019
21.4.2	Appending new categories	1021
21.4.3	Removing categories	1021
21.4.4	Removing unused categories	1021
21.4.5	Setting categories	1022
21.5	Sorting and Order	1023
21.5.1	Reordering	1024
21.5.2	Multi Column Sorting	1025
21.6	Comparisons	1026
21.7	Operations	1028
21.8	Data munging	1029
21.8.1	Getting	1029
21.8.2	String and datetime accessors	1031
21.8.3	Setting	1032
21.8.4	Merging	1034
21.8.5	Unioning	1034
21.8.6	Concatenation	1036
21.9	Getting Data In/Out	1037
21.10	Missing Data	1039
21.11	Differences to R's <i>factor</i>	1040
21.12	Gotchas	1040
21.12.1	Memory Usage	1040
21.12.2	<i>Categorical</i> is not a <i>numpy</i> array	1041
21.12.3	dtype in apply	1042
21.12.4	Categorical Index	1042
21.12.5	Side Effects	1043

<b>22</b>	<b>Visualization</b>	<b>1045</b>
22.1	Basic Plotting: <code>plot</code>	1045
22.2	Other Plots	1048
22.2.1	Bar plots	1050
22.2.2	Histograms	1053
22.2.3	Box Plots	1059
22.2.4	Area Plot	1067
22.2.5	Scatter Plot	1069
22.2.6	Hexagonal Bin Plot	1073
22.2.7	Pie plot	1075
22.3	Plotting with Missing Data	1079
22.4	Plotting Tools	1080
22.4.1	Scatter Matrix Plot	1080
22.4.2	Density Plot	1081
22.4.3	Andrews Curves	1082
22.4.4	Parallel Coordinates	1083
22.4.5	Lag Plot	1084
22.4.6	Autocorrelation Plot	1085
22.4.7	Bootstrap Plot	1086
22.4.8	RadViz	1087
22.5	Plot Formatting	1088
22.5.1	Setting the plot style	1088
22.5.2	General plot style arguments	1088
22.5.3	Controlling the Legend	1089
22.5.4	Scales	1090
22.5.5	Plotting on a Secondary Y-axis	1091
22.5.6	Suppressing Tick Resolution Adjustment	1094
22.5.7	Automatic Date Tick Adjustment	1097
22.5.8	Subplots	1097
22.5.9	Using Layout and Targeting Multiple Axes	1098
22.5.10	Plotting With Error Bars	1101
22.5.11	Plotting Tables	1103
22.5.12	Colormaps	1106
22.6	Plotting directly with <code>matplotlib</code>	1111
22.7	Trellis plotting interface	1112
<b>23</b>	<b>Styling</b>	<b>1113</b>
23.1	Building Styles	1113
23.1.1	Building Styles Summary	1115
23.2	Finer Control: Slicing	1116
23.3	Finer Control: Display Values	1116
23.4	Builtin Styles	1117
23.4.1	Bar charts	1117
23.5	Sharing Styles	1119
23.6	Other Options	1119
23.6.1	Precision	1119
23.6.2	Captions	1120
23.6.3	Table Styles	1120
23.6.4	Hiding the Index or Columns	1120
23.6.5	CSS Classes	1121
23.6.6	Limitations	1121
23.6.7	Terms	1121
23.7	Fun stuff	1121
23.8	Export to Excel	1122

23.9	Extensibility . . . . .	1123
23.9.1	Subclassing . . . . .	1123
<b>24</b>	<b>IO Tools (Text, CSV, HDF5, ...)</b>	<b>1125</b>
24.1	CSV & Text files . . . . .	1125
24.1.1	Parsing options . . . . .	1125
24.1.1.1	Basic . . . . .	1126
24.1.1.2	Column and Index Locations and Names . . . . .	1126
24.1.1.3	General Parsing Configuration . . . . .	1127
24.1.1.4	NA and Missing Data Handling . . . . .	1128
24.1.1.5	Datetime Handling . . . . .	1128
24.1.1.6	Iteration . . . . .	1129
24.1.1.7	Quoting, Compression, and File Format . . . . .	1129
24.1.1.8	Error Handling . . . . .	1130
24.1.2	Specifying column data types . . . . .	1130
24.1.3	Specifying Categorical dtype . . . . .	1132
24.1.4	Naming and Using Columns . . . . .	1134
24.1.4.1	Handling column names . . . . .	1134
24.1.5	Duplicate names parsing . . . . .	1135
24.1.5.1	Filtering columns (usecols) . . . . .	1136
24.1.6	Comments and Empty Lines . . . . .	1136
24.1.6.1	Ignoring line comments and empty lines . . . . .	1136
24.1.6.2	Comments . . . . .	1138
24.1.7	Dealing with Unicode Data . . . . .	1139
24.1.8	Index columns and trailing delimiters . . . . .	1139
24.1.9	Date Handling . . . . .	1140
24.1.9.1	Specifying Date Columns . . . . .	1140
24.1.9.2	Date Parsing Functions . . . . .	1143
24.1.9.3	Inferring Datetime Format . . . . .	1143
24.1.9.4	International Date Formats . . . . .	1144
24.1.10	Specifying method for floating-point conversion . . . . .	1145
24.1.11	Thousand Separators . . . . .	1145
24.1.12	NA Values . . . . .	1146
24.1.13	Infinity . . . . .	1146
24.1.14	Returning Series . . . . .	1146
24.1.15	Boolean values . . . . .	1147
24.1.16	Handling “bad” lines . . . . .	1147
24.1.17	Dialect . . . . .	1148
24.1.18	Quoting and Escape Characters . . . . .	1149
24.1.19	Files with Fixed Width Columns . . . . .	1149
24.1.20	Indexes . . . . .	1151
24.1.20.1	Files with an “implicit” index column . . . . .	1151
24.1.20.2	Reading an index with a MultiIndex . . . . .	1151
24.1.20.3	Reading columns with a MultiIndex . . . . .	1152
24.1.21	Automatically “sniffing” the delimiter . . . . .	1153
24.1.22	Reading multiple files to create a single DataFrame . . . . .	1154
24.1.23	Iterating through files chunk by chunk . . . . .	1154
24.1.24	Specifying the parser engine . . . . .	1156
24.1.25	Reading remote files . . . . .	1156
24.1.26	Writing out Data . . . . .	1156
24.1.26.1	Writing to CSV format . . . . .	1156
24.1.26.2	Writing a formatted string . . . . .	1157
24.2	JSON . . . . .	1157
24.2.1	Writing JSON . . . . .	1157

	24.2.1.1	Orient Options	1158
	24.2.1.2	Date Handling	1160
	24.2.1.3	Fallback Behavior	1161
	24.2.2	Reading JSON	1161
	24.2.2.1	Data Conversion	1162
	24.2.2.2	The Numpy Parameter	1165
	24.2.3	Normalization	1166
	24.2.4	Line delimited json	1167
	24.2.5	Table Schema	1168
24.3	HTML		1172
	24.3.1	Reading HTML Content	1172
	24.3.2	Writing to HTML files	1176
	24.3.3	HTML Table Parsing Gotchas	1179
24.4	Excel files		1180
	24.4.1	Reading Excel Files	1180
	24.4.1.1	ExcelFile class	1181
	24.4.1.2	Specifying Sheets	1181
	24.4.1.3	Reading a MultiIndex	1182
	24.4.1.4	Parsing Specific Columns	1183
	24.4.1.5	Parsing Dates	1183
	24.4.1.6	Cell Converters	1184
	24.4.1.7	dtype Specifications	1184
	24.4.2	Writing Excel Files	1184
	24.4.2.1	Writing Excel Files to Disk	1184
	24.4.2.2	Writing Excel Files to Memory	1185
	24.4.3	Excel writer engines	1185
	24.4.4	Style and Formatting	1186
24.5	Clipboard		1186
24.6	Pickling		1187
	24.6.1	Compressed pickle files	1188
24.7	msgpack		1190
	24.7.1	Read/Write API	1192
24.8	HDF5 (PyTables)		1193
	24.8.1	Read/Write API	1195
	24.8.2	Fixed Format	1196
	24.8.3	Table Format	1197
	24.8.4	Hierarchical Keys	1197
	24.8.5	Storing Types	1199
	24.8.5.1	Storing Mixed Types in a Table	1199
	24.8.5.2	Storing Multi-Index DataFrames	1200
	24.8.6	Querying	1201
	24.8.6.1	Querying a Table	1201
	24.8.6.2	Using timedelta64[ns]	1204
	24.8.6.3	Indexing	1205
	24.8.6.4	Query via Data Columns	1206
	24.8.6.5	Iterator	1208
	24.8.6.6	Advanced Queries	1209
	24.8.6.7	Multiple Table Queries	1211
	24.8.7	Delete from a Table	1213
	24.8.8	Notes & Caveats	1213
	24.8.8.1	Compression	1213
	24.8.8.2	ptrepack	1214
	24.8.8.3	Caveats	1215
	24.8.9	DataTypes	1215

24.8.9.1	Categorical Data	1215
24.8.9.2	String Columns	1216
24.8.10	External Compatibility	1218
24.8.11	Performance	1219
24.9	Feather	1220
24.10	Parquet	1222
24.11	SQL Queries	1223
24.11.1	pandas.read_sql_table	1224
24.11.2	pandas.read_sql_query	1225
24.11.3	pandas.read_sql	1226
24.11.4	pandas.DataFrame.to_sql	1227
24.11.5	Writing DataFrames	1229
24.11.6	Reading Tables	1230
24.11.7	Schema support	1231
24.11.8	Querying	1231
24.11.9	Engine connection examples	1232
24.11.10	Advanced SQLAlchemy queries	1233
24.11.11	Sqlite fallback	1234
24.12	Google BigQuery	1234
24.13	Stata Format	1234
24.13.1	Writing to Stata format	1234
24.13.2	Reading from Stata format	1235
24.13.2.1	Categorical Data	1236
24.14	SAS Formats	1236
24.15	Other file formats	1237
24.15.1	netCDF	1237
24.16	Performance Considerations	1237
<b>25</b>	<b>Enhancing Performance</b>	<b>1241</b>
25.1	Cython (Writing C extensions for pandas)	1241
25.1.1	Pure python	1241
25.1.2	Plain Cython	1242
25.1.3	Adding type	1243
25.1.4	Using ndarray	1244
25.1.5	More advanced techniques	1245
25.2	Using Numba	1246
25.2.1	Jit	1246
25.2.2	Vectorize	1247
25.2.3	Caveats	1247
25.3	Expression Evaluation via eval()	1248
25.3.1	Supported Syntax	1248
25.3.2	eval() Examples	1249
25.3.3	The DataFrame.eval method	1250
25.3.4	Local Variables	1252
25.3.5	pandas.eval() Parsers	1254
25.3.6	pandas.eval() Backends	1254
25.3.7	pandas.eval() Performance	1255
25.3.8	Technical Minutia Regarding Expression Evaluation	1256
<b>26</b>	<b>Sparse data structures</b>	<b>1257</b>
26.1	SparseArray	1259
26.2	SparseIndex objects	1259
26.3	Sparse Dtypes	1259
26.4	Sparse Calculation	1262

26.5	Interaction with <code>scipy.sparse</code> . . . . .	1262
26.5.1	<code>SparseDataFrame</code> . . . . .	1262
26.5.2	<code>SparseSeries</code> . . . . .	1263
<b>27</b>	<b>Frequently Asked Questions (FAQ)</b>	<b>1267</b>
27.1	<code>DataFrame</code> memory usage . . . . .	1267
27.2	Using If/Truth Statements with <code>pandas</code> . . . . .	1269
27.2.1	Bitwise boolean . . . . .	1269
27.2.2	Using the <code>in</code> operator . . . . .	1270
27.3	<code>NaN</code> , Integer NA values and NA type promotions . . . . .	1270
27.3.1	Choice of NA representation . . . . .	1270
27.3.2	Support for integer NA . . . . .	1271
27.3.3	NA type promotions . . . . .	1271
27.3.4	Why not make NumPy like R? . . . . .	1272
27.4	Differences with NumPy . . . . .	1272
27.5	Thread-safety . . . . .	1272
27.6	Byte-Ordering Issues . . . . .	1272
<b>28</b>	<b>rpy2 / R interface</b>	<b>1275</b>
28.1	Transferring R data sets into Python . . . . .	1275
28.2	Converting DataFrames into R objects . . . . .	1276
<b>29</b>	<b>pandas Ecosystem</b>	<b>1277</b>
29.1	Statistics and Machine Learning . . . . .	1277
29.1.1	<code>Statsmodels</code> . . . . .	1277
29.1.2	<code>sklearn-pandas</code> . . . . .	1277
29.1.3	<code>Featuretools</code> . . . . .	1277
29.2	Visualization . . . . .	1277
29.2.1	<code>Bokeh</code> . . . . .	1277
29.2.2	<code>seaborn</code> . . . . .	1278
29.2.3	<code>yhat/ggplot</code> . . . . .	1278
29.2.4	<code>Vincent</code> . . . . .	1278
29.2.5	<code>IPython Vega</code> . . . . .	1278
29.2.6	<code>Plotly</code> . . . . .	1278
29.2.7	<code>QtPandas</code> . . . . .	1278
29.3	IDE . . . . .	1278
29.3.1	<code>IPython</code> . . . . .	1278
29.3.2	<code>quantopian/qgrid</code> . . . . .	1279
29.3.3	<code>Spyder</code> . . . . .	1279
29.4	API . . . . .	1279
29.4.1	<code>pandas-datareader</code> . . . . .	1279
29.4.2	<code>quandl/Python</code> . . . . .	1279
29.4.3	<code>pydatastream</code> . . . . .	1279
29.4.4	<code>pandaSDMX</code> . . . . .	1280
29.4.5	<code>fredapi</code> . . . . .	1280
29.5	Domain Specific . . . . .	1280
29.5.1	<code>Geopandas</code> . . . . .	1280
29.5.2	<code>xarray</code> . . . . .	1280
29.6	Out-of-core . . . . .	1280
29.6.1	<code>Dask</code> . . . . .	1280
29.6.2	<code>Dask-ML</code> . . . . .	1280
29.6.3	<code>Blaze</code> . . . . .	1280
29.6.4	<code>Odo</code> . . . . .	1281
29.7	Data validation . . . . .	1281

29.7.1	Engarde	1281
29.8	Extension Data Types	1281
29.8.1	cyberpandas	1281
29.9	Accessors	1281
<b>30</b>	<b>Comparison with R / R libraries</b>	<b>1283</b>
30.1	Quick Reference	1283
30.1.1	Querying, Filtering, Sampling	1283
30.1.2	Sorting	1284
30.1.3	Transforming	1284
30.1.4	Grouping and Summarizing	1284
30.2	Base R	1284
30.2.1	Slicing with R's <code>c</code>	1284
30.2.2	<code>aggregate</code>	1286
30.2.3	<code>match / %in%</code>	1287
30.2.4	<code>tapply</code>	1287
30.2.5	<code>subset</code>	1288
30.2.6	<code>with</code>	1288
30.3	<code>plyr</code>	1289
30.3.1	<code>ddply</code>	1290
30.4	<code>reshape / reshape2</code>	1290
30.4.1	<code>melt.array</code>	1290
30.4.2	<code>melt.list</code>	1291
30.4.3	<code>melt.data.frame</code>	1291
30.4.4	<code>cast</code>	1292
30.4.5	<code>factor</code>	1294
<b>31</b>	<b>Comparison with SQL</b>	<b>1295</b>
31.1	<code>SELECT</code>	1295
31.2	<code>WHERE</code>	1296
31.3	<code>GROUP BY</code>	1298
31.4	<code>JOIN</code>	1300
31.4.1	<code>INNER JOIN</code>	1300
31.4.2	<code>LEFT OUTER JOIN</code>	1301
31.4.3	<code>RIGHT JOIN</code>	1301
31.4.4	<code>FULL JOIN</code>	1301
31.5	<code>UNION</code>	1302
31.6	Pandas equivalents for some SQL analytic and aggregate functions	1303
31.6.1	Top N rows with offset	1303
31.6.2	Top N rows per group	1303
31.7	<code>UPDATE</code>	1305
31.8	<code>DELETE</code>	1305
<b>32</b>	<b>Comparison with SAS</b>	<b>1307</b>
32.1	Data Structures	1307
32.1.1	General Terminology Translation	1307
32.1.2	<code>DataFrame / Series</code>	1307
32.1.3	<code>Index</code>	1308
32.2	Data Input / Output	1308
32.2.1	Constructing a <code>DataFrame</code> from Values	1308
32.2.2	Reading External Data	1308
32.2.3	Exporting Data	1309
32.3	Data Operations	1309
32.3.1	Operations on Columns	1309



32.3.2	Filtering . . . . .	1310
32.3.3	If/Then Logic . . . . .	1310
32.3.4	Date Functionality . . . . .	1311
32.3.5	Selection of Columns . . . . .	1312
32.3.6	Sorting by Values . . . . .	1313
32.4	String Processing . . . . .	1313
32.4.1	Length . . . . .	1313
32.4.2	Find . . . . .	1314
32.4.3	Substring . . . . .	1314
32.4.4	Scan . . . . .	1314
32.4.5	Uppercase, Lowercase, and Propcase . . . . .	1315
32.5	Merging . . . . .	1316
32.6	Missing Data . . . . .	1317
32.7	GroupBy . . . . .	1319
32.7.1	Aggregation . . . . .	1319
32.7.2	Transformation . . . . .	1319
32.7.3	By Group Processing . . . . .	1320
32.8	Other Considerations . . . . .	1321
32.8.1	Disk vs Memory . . . . .	1321
32.8.2	Data Interop . . . . .	1321
<b>33</b>	<b>Comparison with Stata</b>	<b>1323</b>
33.1	Data Structures . . . . .	1323
33.1.1	General Terminology Translation . . . . .	1323
33.1.2	DataFrame / Series . . . . .	1323
33.1.3	Index . . . . .	1324
33.2	Data Input / Output . . . . .	1324
33.2.1	Constructing a DataFrame from Values . . . . .	1324
33.2.2	Reading External Data . . . . .	1324
33.2.3	Exporting Data . . . . .	1325
33.3	Data Operations . . . . .	1325
33.3.1	Operations on Columns . . . . .	1325
33.3.2	Filtering . . . . .	1326
33.3.3	If/Then Logic . . . . .	1326
33.3.4	Date Functionality . . . . .	1327
33.3.5	Selection of Columns . . . . .	1327
33.3.6	Sorting by Values . . . . .	1328
33.4	String Processing . . . . .	1329
33.4.1	Finding Length of String . . . . .	1329
33.4.2	Finding Position of Substring . . . . .	1329
33.4.3	Extracting Substring by Position . . . . .	1330
33.4.4	Extracting nth Word . . . . .	1330
33.4.5	Changing Case . . . . .	1331
33.5	Merging . . . . .	1331
33.6	Missing Data . . . . .	1333
33.7	GroupBy . . . . .	1335
33.7.1	Aggregation . . . . .	1335
33.7.2	Transformation . . . . .	1335
33.7.3	By Group Processing . . . . .	1336
33.8	Other Considerations . . . . .	1336
33.8.1	Disk vs Memory . . . . .	1336
<b>34</b>	<b>API Reference</b>	<b>1337</b>
34.1	Input/Output . . . . .	1337

34.1.1	Pickling	1337
34.1.1.1	pandas.read_pickle	1337
34.1.2	Flat File	1338
34.1.2.1	pandas.read_table	1339
34.1.2.2	pandas.read_csv	1344
34.1.2.3	pandas.read_fwf	1349
34.1.2.4	pandas.read_msgpack	1355
34.1.3	Clipboard	1355
34.1.3.1	pandas.read_clipboard	1355
34.1.4	Excel	1355
34.1.4.1	pandas.read_excel	1356
34.1.4.2	pandas.ExcelFile.parse	1359
34.1.5	JSON	1359
34.1.5.1	pandas.read_json	1360
34.1.5.2	pandas.io.json.json_normalize	1363
34.1.5.3	pandas.io.json.build_table_schema	1364
34.1.6	HTML	1365
34.1.6.1	pandas.read_html	1365
34.1.7	HDFStore: PyTables (HDF5)	1368
34.1.7.1	pandas.read_hdf	1368
34.1.7.2	pandas.HDFStore.put	1369
34.1.7.3	pandas.HDFStore.append	1370
34.1.7.4	pandas.HDFStore.get	1370
34.1.7.5	pandas.HDFStore.select	1371
34.1.7.6	pandas.HDFStore.info	1371
34.1.7.7	pandas.HDFStore.keys	1371
34.1.8	Feather	1371
34.1.8.1	pandas.read_feather	1371
34.1.9	Parquet	1372
34.1.9.1	pandas.read_parquet	1372
34.1.10	SAS	1372
34.1.10.1	pandas.read_sas	1372
34.1.11	SQL	1373
34.1.12	Google BigQuery	1373
34.1.12.1	pandas.read_gbq	1373
34.1.13	STATA	1374
34.1.13.1	pandas.read_stata	1374
34.1.13.2	pandas.io.stata.StataReader.data	1376
34.1.13.3	pandas.io.stata.StataReader.data_label	1377
34.1.13.4	pandas.io.stata.StataReader.value_labels	1377
34.1.13.5	pandas.io.stata.StataReader.variable_labels	1377
34.1.13.6	pandas.io.stata.StataWriter.write_file	1377
34.2	General functions	1377
34.2.1	Data manipulations	1377
34.2.1.1	pandas.melt	1378
34.2.1.2	pandas.pivot	1379
34.2.1.3	pandas.pivot_table	1380
34.2.1.4	pandas.crosstab	1382
34.2.1.5	pandas.cut	1383
34.2.1.6	pandas.qcut	1386
34.2.1.7	pandas.merge	1387
34.2.1.8	pandas.merge_ordered	1389
34.2.1.9	pandas.merge_asof	1391
34.2.1.10	pandas.concat	1394

34.2.1.11	pandas.get_dummies	1398
34.2.1.12	pandas.factorize	1400
34.2.1.13	pandas.unique	1401
34.2.1.14	pandas.wide_to_long	1403
34.2.2	Top-level missing data	1407
34.2.2.1	pandas.isna	1407
34.2.2.2	pandas.isnull	1408
34.2.2.3	pandas.notna	1409
34.2.2.4	pandas.notnull	1411
34.2.3	Top-level conversions	1412
34.2.3.1	pandas.to_numeric	1412
34.2.4	Top-level dealing with datetimelike	1414
34.2.4.1	pandas.to_datetime	1414
34.2.4.2	pandas.to_timedelta	1417
34.2.4.3	pandas.date_range	1418
34.2.4.4	pandas.bdate_range	1420
34.2.4.5	pandas.period_range	1421
34.2.4.6	pandas.timedelta_range	1422
34.2.4.7	pandas.infer_freq	1423
34.2.5	Top-level dealing with intervals	1424
34.2.5.1	pandas.interval_range	1424
34.2.6	Top-level evaluation	1425
34.2.6.1	pandas.eval	1425
34.2.7	Testing	1427
34.2.7.1	pandas.test	1427
34.3	Series	1427
34.3.1	Constructor	1427
34.3.1.1	pandas.Series	1427
34.3.2	Attributes	1630
34.3.2.1	pandas.Series.empty	1631
34.3.2.2	pandas.Series.is_copy	1631
34.3.2.3	pandas.Series.name	1631
34.3.3	Conversion	1631
34.3.4	Indexing, iteration	1631
34.3.4.1	pandas.Series.__iter__	1632
34.3.5	Binary operator functions	1632
34.3.6	Function application, GroupBy & Window	1633
34.3.7	Computations / Descriptive Stats	1634
34.3.8	Reindexing / Selection / Label manipulation	1635
34.3.9	Missing data handling	1636
34.3.10	Reshaping, sorting	1636
34.3.11	Combining / joining / merging	1637
34.3.12	Time series-related	1637
34.3.13	Datetimelike Properties	1637
34.3.13.1	pandas.Series.dt.date	1638
34.3.13.2	pandas.Series.dt.time	1638
34.3.13.3	pandas.Series.dt.year	1638
34.3.13.4	pandas.Series.dt.month	1638
34.3.13.5	pandas.Series.dt.day	1638
34.3.13.6	pandas.Series.dt.hour	1639
34.3.13.7	pandas.Series.dt.minute	1639
34.3.13.8	pandas.Series.dt.second	1639
34.3.13.9	pandas.Series.dt.microsecond	1639
34.3.13.10	pandas.Series.dt.nanosecond	1639

34.3.13.11	pandas.Series.dt.week	1639
34.3.13.12	pandas.Series.dt.weekofyear	1639
34.3.13.13	pandas.Series.dt.dayofweek	1639
34.3.13.14	pandas.Series.dt.weekday	1639
34.3.13.15	pandas.Series.dt.dayofyear	1640
34.3.13.16	pandas.Series.dt.quarter	1640
34.3.13.17	pandas.Series.dt.is_month_start	1640
34.3.13.18	pandas.Series.dt.is_month_end	1640
34.3.13.19	pandas.Series.dt.is_quarter_start	1641
34.3.13.20	pandas.Series.dt.is_quarter_end	1641
34.3.13.21	pandas.Series.dt.is_year_start	1642
34.3.13.22	pandas.Series.dt.is_year_end	1643
34.3.13.23	pandas.Series.dt.is_leap_year	1643
34.3.13.24	pandas.Series.dt.daysinmonth	1644
34.3.13.25	pandas.Series.dt.days_in_month	1644
34.3.13.26	pandas.Series.dt.tz	1644
34.3.13.27	pandas.Series.dt.freq	1644
34.3.13.28	pandas.Series.dt.to_period	1645
34.3.13.29	pandas.Series.dt.to_pydatetime	1646
34.3.13.30	pandas.Series.dt.tz_localize	1646
34.3.13.31	pandas.Series.dt.tz_convert	1648
34.3.13.32	pandas.Series.dt.normalize	1649
34.3.13.33	pandas.Series.dt.strftime	1649
34.3.13.34	pandas.Series.dt.round	1650
34.3.13.35	pandas.Series.dt.floor	1651
34.3.13.36	pandas.Series.dt.ceil	1652
34.3.13.37	pandas.Series.dt.month_name	1652
34.3.13.38	pandas.Series.dt.day_name	1653
34.3.13.39	pandas.Series.dt.days	1653
34.3.13.40	pandas.Series.dt.seconds	1653
34.3.13.41	pandas.Series.dt.microseconds	1653
34.3.13.42	pandas.Series.dt.nanoseconds	1653
34.3.13.43	pandas.Series.dt.components	1653
34.3.13.44	pandas.Series.dt.to_pytimedelta	1654
34.3.13.45	pandas.Series.dt.total_seconds	1654
34.3.14	String handling	1655
34.3.14.1	pandas.Series.str.capitalize	1657
34.3.14.2	pandas.Series.str.cat	1659
34.3.14.3	pandas.Series.str.center	1661
34.3.14.4	pandas.Series.str.contains	1661
34.3.14.5	pandas.Series.str.count	1663
34.3.14.6	pandas.Series.str.decode	1664
34.3.14.7	pandas.Series.str.encode	1665
34.3.14.8	pandas.Series.str.endswith	1665
34.3.14.9	pandas.Series.str.extract	1666
34.3.14.10	pandas.Series.str.extractall	1667
34.3.14.11	pandas.Series.str.find	1669
34.3.14.12	pandas.Series.str.findall	1669
34.3.14.13	pandas.Series.str.get	1670
34.3.14.14	pandas.Series.str.index	1671
34.3.14.15	pandas.Series.str.join	1672
34.3.14.16	pandas.Series.str.len	1673
34.3.14.17	pandas.Series.str.ljust	1673
34.3.14.18	pandas.Series.str.lower	1673

34.3.14.19	pandas.Series.str.lstrip . . . . .	1674
34.3.14.20	pandas.Series.str.match . . . . .	1675
34.3.14.21	pandas.Series.str.normalize . . . . .	1675
34.3.14.22	pandas.Series.str.pad . . . . .	1675
34.3.14.23	pandas.Series.str.partition . . . . .	1676
34.3.14.24	pandas.Series.str.repeat . . . . .	1676
34.3.14.25	pandas.Series.str.replace . . . . .	1677
34.3.14.26	pandas.Series.str.rfind . . . . .	1679
34.3.14.27	pandas.Series.str.rindex . . . . .	1679
34.3.14.28	pandas.Series.str.rjust . . . . .	1679
34.3.14.29	pandas.Series.str.rpartition . . . . .	1680
34.3.14.30	pandas.Series.str.rstrip . . . . .	1681
34.3.14.31	pandas.Series.str.slice . . . . .	1681
34.3.14.32	pandas.Series.str.slice_replace . . . . .	1681
34.3.14.33	pandas.Series.str.split . . . . .	1682
34.3.14.34	pandas.Series.str.split . . . . .	1684
34.3.14.35	pandas.Series.str.startswith . . . . .	1684
34.3.14.36	pandas.Series.str.strip . . . . .	1685
34.3.14.37	pandas.Series.str.swapcase . . . . .	1686
34.3.14.38	pandas.Series.str.title . . . . .	1687
34.3.14.39	pandas.Series.str.translate . . . . .	1688
34.3.14.40	pandas.Series.str.upper . . . . .	1688
34.3.14.41	pandas.Series.str.wrap . . . . .	1690
34.3.14.42	pandas.Series.str.zfill . . . . .	1691
34.3.14.43	pandas.Series.str.isalnum . . . . .	1691
34.3.14.44	pandas.Series.str.isalpha . . . . .	1691
34.3.14.45	pandas.Series.str.isdigit . . . . .	1691
34.3.14.46	pandas.Series.str.isspace . . . . .	1691
34.3.14.47	pandas.Series.str.islower . . . . .	1692
34.3.14.48	pandas.Series.str.isupper . . . . .	1692
34.3.14.49	pandas.Series.str.istitle . . . . .	1692
34.3.14.50	pandas.Series.str.isnumeric . . . . .	1692
34.3.14.51	pandas.Series.str.isdecimal . . . . .	1692
34.3.14.52	pandas.Series.str.get_dummies . . . . .	1692
34.3.15	Categorical . . . . .	1693
34.3.15.1	pandas.api.types.CategoricalDtype . . . . .	1693
34.3.15.2	pandas.Categorical . . . . .	1694
34.3.15.3	pandas.Series.cat.categories . . . . .	1698
34.3.15.4	pandas.Series.cat.ordered . . . . .	1698
34.3.15.5	pandas.Series.cat.codes . . . . .	1698
34.3.15.6	pandas.Series.cat.rename_categories . . . . .	1699
34.3.15.7	pandas.Series.cat.reorder_categories . . . . .	1700
34.3.15.8	pandas.Series.cat.add_categories . . . . .	1700
34.3.15.9	pandas.Series.cat.remove_categories . . . . .	1701
34.3.15.10	pandas.Series.cat.remove_unused_categories . . . . .	1701
34.3.15.11	pandas.Series.cat.set_categories . . . . .	1702
34.3.15.12	pandas.Series.cat.as_ordered . . . . .	1702
34.3.15.13	pandas.Series.cat.as_unordered . . . . .	1703
34.3.16	Plotting . . . . .	1703
34.3.16.1	pandas.Series.plot.area . . . . .	1703
34.3.16.2	pandas.Series.plot.bar . . . . .	1703
34.3.16.3	pandas.Series.plot.barh . . . . .	1704
34.3.16.4	pandas.Series.plot.box . . . . .	1704
34.3.16.5	pandas.Series.plot.density . . . . .	1704

34.3.16.6	pandas.Series.plot.hist	1705
34.3.16.7	pandas.Series.plot.kde	1705
34.3.16.8	pandas.Series.plot.line	1706
34.3.16.9	pandas.Series.plot.pie	1706
34.3.17	Serialization / IO / Conversion	1707
34.3.18	Sparse	1707
34.3.18.1	pandas.SparseSeries.to_coo	1707
34.3.18.2	pandas.SparseSeries.from_coo	1708
34.4	DataFrame	1709
34.4.1	Constructor	1709
34.4.1.1	pandas.DataFrame	1709
34.4.2	Attributes and underlying data	1942
34.4.2.1	pandas.DataFrame.is_copy	1943
34.4.3	Conversion	1943
34.4.4	Indexing, iteration	1943
34.4.4.1	pandas.DataFrame.__iter__	1944
34.4.5	Binary operator functions	1944
34.4.6	Function application, GroupBy & Window	1945
34.4.7	Computations / Descriptive Stats	1945
34.4.8	Reindexing / Selection / Label manipulation	1947
34.4.9	Missing data handling	1948
34.4.10	Reshaping, sorting, transposing	1948
34.4.11	Combining / joining / merging	1949
34.4.12	Time series-related	1949
34.4.13	Plotting	1949
34.4.13.1	pandas.DataFrame.plot.area	1950
34.4.13.2	pandas.DataFrame.plot.bar	1950
34.4.13.3	pandas.DataFrame.plot.barh	1951
34.4.13.4	pandas.DataFrame.plot.box	1953
34.4.13.5	pandas.DataFrame.plot.density	1953
34.4.13.6	pandas.DataFrame.plot.hexbin	1954
34.4.13.7	pandas.DataFrame.plot.hist	1956
34.4.13.8	pandas.DataFrame.plot.kde	1956
34.4.13.9	pandas.DataFrame.plot.line	1957
34.4.13.10	pandas.DataFrame.plot.pie	1958
34.4.13.11	pandas.DataFrame.plot.scatter	1959
34.4.14	Serialization / IO / Conversion	1960
34.4.15	Sparse	1961
34.4.15.1	pandas.SparseDataFrame.to_coo	1961
34.5	Panel	1962
34.5.1	Constructor	1962
34.5.1.1	pandas.Panel	1962
34.5.2	Attributes and underlying data	2096
34.5.3	Conversion	2096
34.5.4	Getting and setting	2096
34.5.5	Indexing, iteration, slicing	2096
34.5.5.1	pandas.Panel.__iter__	2097
34.5.6	Binary operator functions	2097
34.5.7	Function application, GroupBy	2098
34.5.8	Computations / Descriptive Stats	2098
34.5.9	Reindexing / Selection / Label manipulation	2099
34.5.9.1	pandas.Panel.drop	2099
34.5.10	Missing data handling	2099
34.5.11	Reshaping, sorting, transposing	2099

34.5.12	Combining / joining / merging	2100
34.5.13	Time series-related	2100
34.5.14	Serialization / IO / Conversion	2100
34.6	Index	2100
34.6.1	pandas.Index	2100
34.6.1.1	pandas.Index.T	2102
34.6.1.2	pandas.Index.base	2102
34.6.1.3	pandas.Index.data	2102
34.6.1.4	pandas.Index.dtype	2102
34.6.1.5	pandas.Index.dtype_str	2102
34.6.1.6	pandas.Index.flags	2102
34.6.1.7	pandas.Index.hasnans	2103
34.6.1.8	pandas.Index.inferred_type	2103
34.6.1.9	pandas.Index.is_monotonic	2103
34.6.1.10	pandas.Index.is_monotonic_decreasing	2103
34.6.1.11	pandas.Index.is_monotonic_increasing	2103
34.6.1.12	pandas.Index.is_unique	2103
34.6.1.13	pandas.Index.itemsize	2104
34.6.1.14	pandas.Index.nbytes	2104
34.6.1.15	pandas.Index.ndim	2104
34.6.1.16	pandas.Index.shape	2104
34.6.1.17	pandas.Index.size	2104
34.6.1.18	pandas.Index.strides	2104
34.6.1.19	pandas.Index.values	2104
34.6.1.20	pandas.Index.all	2107
34.6.1.21	pandas.Index.any	2108
34.6.1.22	pandas.Index.append	2108
34.6.1.23	pandas.Index.argmax	2108
34.6.1.24	pandas.Index.argmin	2109
34.6.1.25	pandas.Index.argsort	2109
34.6.1.26	pandas.Index.asof	2109
34.6.1.27	pandas.Index.asof_locs	2110
34.6.1.28	pandas.Index.astype	2110
34.6.1.29	pandas.Index.contains	2110
34.6.1.30	pandas.Index.copy	2110
34.6.1.31	pandas.Index.delete	2111
34.6.1.32	pandas.Index.difference	2111
34.6.1.33	pandas.Index.drop	2111
34.6.1.34	pandas.Index.drop_duplicates	2112
34.6.1.35	pandas.Index.dropna	2112
34.6.1.36	pandas.Index.duplicated	2113
34.6.1.37	pandas.Index.equals	2113
34.6.1.38	pandas.Index.factorize	2114
34.6.1.39	pandas.Index.fillna	2115
34.6.1.40	pandas.Index.format	2115
34.6.1.41	pandas.Index.get_duplicates	2116
34.6.1.42	pandas.Index.get_indexer	2116
34.6.1.43	pandas.Index.get_indexer_for	2117
34.6.1.44	pandas.Index.get_indexer_non_unique	2117
34.6.1.45	pandas.Index.get_level_values	2118
34.6.1.46	pandas.Index.get_loc	2118
34.6.1.47	pandas.Index.get_slice_bound	2119
34.6.1.48	pandas.Index.get_value	2119
34.6.1.49	pandas.Index.get_values	2119

34.6.1.50	pandas.Index.groupby	2120
34.6.1.51	pandas.Index.identical	2120
34.6.1.52	pandas.Index.insert	2120
34.6.1.53	pandas.Index.intersection	2121
34.6.1.54	pandas.Index.is_	2121
34.6.1.55	pandas.Index.is_categorical	2121
34.6.1.56	pandas.Index.isin	2122
34.6.1.57	pandas.Index.isna	2123
34.6.1.58	pandas.Index.isnull	2124
34.6.1.59	pandas.Index.item	2125
34.6.1.60	pandas.Index.join	2125
34.6.1.61	pandas.Index.map	2126
34.6.1.62	pandas.Index.max	2126
34.6.1.63	pandas.Index.memory_usage	2127
34.6.1.64	pandas.Index.min	2127
34.6.1.65	pandas.Index.notna	2128
34.6.1.66	pandas.Index.notnull	2128
34.6.1.67	pandas.Index.nunique	2129
34.6.1.68	pandas.Index.putmask	2129
34.6.1.69	pandas.Index.ravel	2129
34.6.1.70	pandas.Index.reindex	2130
34.6.1.71	pandas.Index.rename	2130
34.6.1.72	pandas.Index.repeat	2130
34.6.1.73	pandas.Index.searchsorted	2131
34.6.1.74	pandas.Index.set_names	2132
34.6.1.75	pandas.Index.set_value	2133
34.6.1.76	pandas.Index.shift	2133
34.6.1.77	pandas.Index.slice_indexer	2134
34.6.1.78	pandas.Index.slice_locs	2135
34.6.1.79	pandas.Index.sort_values	2135
34.6.1.80	pandas.Index.sortlevel	2136
34.6.1.81	pandas.Index.str	2136
34.6.1.82	pandas.Index.summary	2137
34.6.1.83	pandas.Index.symmetric_difference	2137
34.6.1.84	pandas.Index.take	2137
34.6.1.85	pandas.Index.to_frame	2138
34.6.1.86	pandas.Index.to_native_types	2138
34.6.1.87	pandas.Index.to_series	2139
34.6.1.88	pandas.Index.tolist	2139
34.6.1.89	pandas.Index.transpose	2139
34.6.1.90	pandas.Index.union	2139
34.6.1.91	pandas.Index.unique	2140
34.6.1.92	pandas.Index.value_counts	2140
34.6.1.93	pandas.Index.where	2141
34.6.2	Attributes	2141
34.6.2.1	pandas.Index.has_duplicates	2142
34.6.2.2	pandas.Index.is_all_dates	2142
34.6.2.3	pandas.Index.name	2142
34.6.2.4	pandas.Index.names	2142
34.6.2.5	pandas.Index.empty	2142
34.6.3	Modifying and Computations	2142
34.6.3.1	pandas.Index.is_boolean	2143
34.6.3.2	pandas.Index.is_floating	2143
34.6.3.3	pandas.Index.is_integer	2143



34.6.3.4	pandas.Index.is_interval	2143
34.6.3.5	pandas.Index.is_lexsorted_for_tuple	2143
34.6.3.6	pandas.Index.is_mixed	2143
34.6.3.7	pandas.Index.is_numeric	2144
34.6.3.8	pandas.Index.is_object	2144
34.6.4	Missing Values	2144
34.6.5	Conversion	2144
34.6.5.1	pandas.Index.view	2144
34.6.6	Sorting	2144
34.6.7	Time-specific operations	2144
34.6.8	Combining / joining / set operations	2145
34.6.9	Selecting	2145
34.7	Numeric Index	2145
34.7.1	pandas.RangeIndex	2146
34.7.1.1	pandas.RangeIndex.from_range	2147
34.7.2	pandas.Int64Index	2147
34.7.3	pandas.UInt64Index	2147
34.7.4	pandas.Float64Index	2148
34.8	CategoricalIndex	2149
34.8.1	pandas.CategoricalIndex	2149
34.8.1.1	pandas.CategoricalIndex.rename_categories	2150
34.8.1.2	pandas.CategoricalIndex.reorder_categories	2151
34.8.1.3	pandas.CategoricalIndex.add_categories	2151
34.8.1.4	pandas.CategoricalIndex.remove_categories	2152
34.8.1.5	pandas.CategoricalIndex.remove_unused_categories	2152
34.8.1.6	pandas.CategoricalIndex.set_categories	2153
34.8.1.7	pandas.CategoricalIndex.as_ordered	2153
34.8.1.8	pandas.CategoricalIndex.as_unordered	2154
34.8.1.9	pandas.CategoricalIndex.map	2154
34.8.2	Categorical Components	2155
34.8.2.1	pandas.CategoricalIndex.codes	2155
34.8.2.2	pandas.CategoricalIndex.categories	2155
34.8.2.3	pandas.CategoricalIndex.ordered	2156
34.9	IntervalIndex	2156
34.9.1	pandas.IntervalIndex	2156
34.9.1.1	pandas.IntervalIndex.closed	2157
34.9.1.2	pandas.IntervalIndex.is_non_overlapping_monotonic	2157
34.9.1.3	pandas.IntervalIndex.left	2157
34.9.1.4	pandas.IntervalIndex.length	2158
34.9.1.5	pandas.IntervalIndex.mid	2158
34.9.1.6	pandas.IntervalIndex.right	2158
34.9.1.7	pandas.IntervalIndex.values	2158
34.9.1.8	pandas.IntervalIndex.contains	2158
34.9.1.9	pandas.IntervalIndex.from_arrays	2159
34.9.1.10	pandas.IntervalIndex.from_breaks	2160
34.9.1.11	pandas.IntervalIndex.from_tuples	2161
34.9.1.12	pandas.IntervalIndex.get_indexer	2161
34.9.1.13	pandas.IntervalIndex.get_loc	2162
34.9.2	IntervalIndex Components	2163
34.10	MultiIndex	2164
34.10.1	pandas.MultiIndex	2164
34.10.1.1	pandas.MultiIndex.names	2165
34.10.1.2	pandas.MultiIndex.nlevels	2165
34.10.1.3	pandas.MultiIndex.levshape	2165

34.10.1.4	pandas.MultiIndex.from_arrays	2166
34.10.1.5	pandas.MultiIndex.from_tuples	2166
34.10.1.6	pandas.MultiIndex.from_product	2167
34.10.1.7	pandas.MultiIndex.set_levels	2167
34.10.1.8	pandas.MultiIndex.set_labels	2168
34.10.1.9	pandas.MultiIndex.to_hierarchical	2169
34.10.1.10	pandas.MultiIndex.to_frame	2170
34.10.1.11	pandas.MultiIndex.is_lexsorted	2170
34.10.1.12	pandas.MultiIndex.sortlevel	2170
34.10.1.13	pandas.MultiIndex.droplevel	2170
34.10.1.14	pandas.MultiIndex.swaplevel	2171
34.10.1.15	pandas.MultiIndex.reorder_levels	2171
34.10.1.16	pandas.MultiIndex.remove_unused_levels	2172
34.10.2	pandas.IndexSlice	2172
34.10.3	MultiIndex Constructors	2173
34.10.4	MultiIndex Attributes	2173
34.10.4.1	pandas.MultiIndex.levels	2173
34.10.4.2	pandas.MultiIndex.labels	2173
34.10.5	MultiIndex Components	2173
34.10.5.1	pandas.MultiIndex.unique	2174
34.10.6	MultiIndex Selecting	2174
34.10.6.1	pandas.MultiIndex.get_loc	2174
34.10.6.2	pandas.MultiIndex.get_indexer	2175
34.10.6.3	pandas.MultiIndex.get_level_values	2176
34.11	DatetimeIndex	2176
34.11.1	pandas.DatetimeIndex	2176
34.11.1.1	pandas.DatetimeIndex.year	2178
34.11.1.2	pandas.DatetimeIndex.month	2179
34.11.1.3	pandas.DatetimeIndex.day	2179
34.11.1.4	pandas.DatetimeIndex.hour	2179
34.11.1.5	pandas.DatetimeIndex.minute	2179
34.11.1.6	pandas.DatetimeIndex.second	2179
34.11.1.7	pandas.DatetimeIndex.microsecond	2179
34.11.1.8	pandas.DatetimeIndex.nanosecond	2179
34.11.1.9	pandas.DatetimeIndex.date	2179
34.11.1.10	pandas.DatetimeIndex.time	2179
34.11.1.11	pandas.DatetimeIndex.dayofyear	2180
34.11.1.12	pandas.DatetimeIndex.weekofyear	2180
34.11.1.13	pandas.DatetimeIndex.week	2180
34.11.1.14	pandas.DatetimeIndex.dayofweek	2180
34.11.1.15	pandas.DatetimeIndex.weekday	2180
34.11.1.16	pandas.DatetimeIndex.quarter	2180
34.11.1.17	pandas.DatetimeIndex.freq	2180
34.11.1.18	pandas.DatetimeIndex.freqstr	2180
34.11.1.19	pandas.DatetimeIndex.is_month_start	2180
34.11.1.20	pandas.DatetimeIndex.is_month_end	2181
34.11.1.21	pandas.DatetimeIndex.is_quarter_start	2181
34.11.1.22	pandas.DatetimeIndex.is_quarter_end	2182
34.11.1.23	pandas.DatetimeIndex.is_year_start	2183
34.11.1.24	pandas.DatetimeIndex.is_year_end	2183
34.11.1.25	pandas.DatetimeIndex.is_leap_year	2184
34.11.1.26	pandas.DatetimeIndex.inferred_freq	2185
34.11.1.27	pandas.DatetimeIndex.normalize	2186
34.11.1.28	pandas.DatetimeIndex.strftime	2186

34.11.1.29	pandas.DatetimeIndex.snap	2187
34.11.1.30	pandas.DatetimeIndex.tz_convert	2187
34.11.1.31	pandas.DatetimeIndex.tz_localize	2188
34.11.1.32	pandas.DatetimeIndex.round	2189
34.11.1.33	pandas.DatetimeIndex.floor	2190
34.11.1.34	pandas.DatetimeIndex.ceil	2191
34.11.1.35	pandas.DatetimeIndex.to_period	2192
34.11.1.36	pandas.DatetimeIndex.to_perioddelta	2192
34.11.1.37	pandas.DatetimeIndex.to_pydatetime	2193
34.11.1.38	pandas.DatetimeIndex.to_series	2193
34.11.1.39	pandas.DatetimeIndex.to_frame	2193
34.11.1.40	pandas.DatetimeIndex.month_name	2194
34.11.1.41	pandas.DatetimeIndex.day_name	2194
34.11.2	Time/Date Components	2194
34.11.2.1	pandas.DatetimeIndex.tz	2195
34.11.3	Selecting	2195
34.11.3.1	pandas.DatetimeIndex.indexer_at_time	2195
34.11.3.2	pandas.DatetimeIndex.indexer_between_time	2196
34.11.4	Time-specific operations	2196
34.11.5	Conversion	2196
34.12	TimedeltaIndex	2197
34.12.1	pandas.TimedeltaIndex	2197
34.12.1.1	pandas.TimedeltaIndex.days	2198
34.12.1.2	pandas.TimedeltaIndex.seconds	2198
34.12.1.3	pandas.TimedeltaIndex.microseconds	2198
34.12.1.4	pandas.TimedeltaIndex.nanoseconds	2198
34.12.1.5	pandas.TimedeltaIndex.components	2199
34.12.1.6	pandas.TimedeltaIndex.inferred_freq	2199
34.12.1.7	pandas.TimedeltaIndex.to_pytimedelta	2199
34.12.1.8	pandas.TimedeltaIndex.to_series	2199
34.12.1.9	pandas.TimedeltaIndex.round	2200
34.12.1.10	pandas.TimedeltaIndex.floor	2200
34.12.1.11	pandas.TimedeltaIndex.ceil	2201
34.12.1.12	pandas.TimedeltaIndex.to_frame	2202
34.12.2	Components	2202
34.12.3	Conversion	2203
34.13	PeriodIndex	2203
34.13.1	pandas.PeriodIndex	2203
34.13.1.1	pandas.PeriodIndex.day	2205
34.13.1.2	pandas.PeriodIndex.dayofweek	2205
34.13.1.3	pandas.PeriodIndex.dayofyear	2205
34.13.1.4	pandas.PeriodIndex.days_in_month	2205
34.13.1.5	pandas.PeriodIndex.daysinmonth	2205
34.13.1.6	pandas.PeriodIndex.freq	2205
34.13.1.7	pandas.PeriodIndex.freqstr	2205
34.13.1.8	pandas.PeriodIndex.hour	2206
34.13.1.9	pandas.PeriodIndex.is_leap_year	2206
34.13.1.10	pandas.PeriodIndex.minute	2206
34.13.1.11	pandas.PeriodIndex.month	2206
34.13.1.12	pandas.PeriodIndex.quarter	2206
34.13.1.13	pandas.PeriodIndex.second	2206
34.13.1.14	pandas.PeriodIndex.week	2206
34.13.1.15	pandas.PeriodIndex.weekday	2206
34.13.1.16	pandas.PeriodIndex.weekofyear	2206

34.13.1.17	pandas.PeriodIndex.year . . . . .	2207
34.13.1.18	pandas.PeriodIndex.asfreq . . . . .	2207
34.13.1.19	pandas.PeriodIndex.strftime . . . . .	2208
34.13.1.20	pandas.PeriodIndex.to_timestamp . . . . .	2208
34.13.1.21	pandas.PeriodIndex.tz_convert . . . . .	2209
34.13.1.22	pandas.PeriodIndex.tz_localize . . . . .	2209
34.13.2	Attributes . . . . .	2209
34.13.2.1	pandas.PeriodIndex.end_time . . . . .	2210
34.13.2.2	pandas.PeriodIndex.qyear . . . . .	2210
34.13.2.3	pandas.PeriodIndex.start_time . . . . .	2210
34.13.3	Methods . . . . .	2210
34.14	Scalars . . . . .	2210
34.14.1	Period . . . . .	2210
34.14.1.1	pandas.Period . . . . .	2210
34.14.2	Attributes . . . . .	2219
34.14.2.1	pandas.Period.end_time . . . . .	2220
34.14.2.2	pandas.Period.freq . . . . .	2220
34.14.2.3	pandas.Period.freqstr . . . . .	2220
34.14.2.4	pandas.Period.is_leap_year . . . . .	2220
34.14.2.5	pandas.Period.month . . . . .	2220
34.14.2.6	pandas.Period.ordinal . . . . .	2220
34.14.2.7	pandas.Period.quarter . . . . .	2220
34.14.2.8	pandas.Period.qyear . . . . .	2220
34.14.2.9	pandas.Period.weekday . . . . .	2220
34.14.2.10	pandas.Period.weekofyear . . . . .	2220
34.14.2.11	pandas.Period.year . . . . .	2221
34.14.3	Methods . . . . .	2221
34.14.3.1	pandas.Period.now . . . . .	2221
34.14.4	Timestamp . . . . .	2221
34.14.4.1	pandas.Timestamp . . . . .	2221
34.14.5	Properties . . . . .	2231
34.14.5.1	pandas.Timestamp.asm8 . . . . .	2231
34.14.5.2	pandas.Timestamp.day . . . . .	2231
34.14.5.3	pandas.Timestamp.dayofweek . . . . .	2231
34.14.5.4	pandas.Timestamp.dayofyear . . . . .	2231
34.14.5.5	pandas.Timestamp.days_in_month . . . . .	2232
34.14.5.6	pandas.Timestamp.daysinmonth . . . . .	2232
34.14.5.7	pandas.Timestamp.fold . . . . .	2232
34.14.5.8	pandas.Timestamp.hour . . . . .	2232
34.14.5.9	pandas.Timestamp.is_leap_year . . . . .	2232
34.14.5.10	pandas.Timestamp.is_month_end . . . . .	2232
34.14.5.11	pandas.Timestamp.is_month_start . . . . .	2232
34.14.5.12	pandas.Timestamp.is_quarter_end . . . . .	2232
34.14.5.13	pandas.Timestamp.is_quarter_start . . . . .	2232
34.14.5.14	pandas.Timestamp.is_year_end . . . . .	2232
34.14.5.15	pandas.Timestamp.is_year_start . . . . .	2232
34.14.5.16	pandas.Timestamp.max . . . . .	2233
34.14.5.17	pandas.Timestamp.microsecond . . . . .	2233
34.14.5.18	pandas.Timestamp.min . . . . .	2233
34.14.5.19	pandas.Timestamp.minute . . . . .	2233
34.14.5.20	pandas.Timestamp.month . . . . .	2233
34.14.5.21	pandas.Timestamp.nanosecond . . . . .	2233
34.14.5.22	pandas.Timestamp.quarter . . . . .	2233
34.14.5.23	pandas.Timestamp.resolution . . . . .	2233

34.14.5.24	pandas.Timestamp.second	2233
34.14.5.25	pandas.Timestamp.tzinfo	2233
34.14.5.26	pandas.Timestamp.value	2233
34.14.5.27	pandas.Timestamp.week	2234
34.14.5.28	pandas.Timestamp.weekofyear	2234
34.14.5.29	pandas.Timestamp.year	2234
34.14.6	Methods	2234
34.14.6.1	pandas.Timestamp.freq	2235
34.14.6.2	pandas.Timestamp.freqstr	2235
34.14.6.3	pandas.Timestamp.isoformat	2235
34.14.7	Interval	2235
34.14.7.1	pandas.Interval	2235
34.14.8	Properties	2239
34.14.9	Timedelta	2239
34.14.9.1	pandas.Timedelta	2239
34.14.10	Properties	2244
34.14.10.1	pandas.Timedelta.freq	2244
34.14.10.2	pandas.Timedelta.is_populated	2244
34.14.10.3	pandas.Timedelta.max	2244
34.14.10.4	pandas.Timedelta.min	2244
34.14.10.5	pandas.Timedelta.value	2244
34.14.11	Methods	2244
34.15	Frequencies	2245
34.15.1	pandas.tseries.frequencies.to_offset	2245
34.16	Window	2246
34.16.1	Standard moving window functions	2246
34.16.1.1	pandas.core.window.Rolling.count	2246
34.16.1.2	pandas.core.window.Rolling.sum	2247
34.16.1.3	pandas.core.window.Rolling.mean	2248
34.16.1.4	pandas.core.window.Rolling.median	2249
34.16.1.5	pandas.core.window.Rolling.var	2250
34.16.1.6	pandas.core.window.Rolling.std	2251
34.16.1.7	pandas.core.window.Rolling.min	2252
34.16.1.8	pandas.core.window.Rolling.max	2253
34.16.1.9	pandas.core.window.Rolling.corr	2253
34.16.1.10	pandas.core.window.Rolling.cov	2253
34.16.1.11	pandas.core.window.Rolling.skew	2254
34.16.1.12	pandas.core.window.Rolling.kurt	2254
34.16.1.13	pandas.core.window.Rolling.apply	2255
34.16.1.14	pandas.core.window.Rolling.aggregate	2255
34.16.1.15	pandas.core.window.Rolling.quantile	2257
34.16.1.16	pandas.core.window.Window.mean	2258
34.16.1.17	pandas.core.window.Window.sum	2259
34.16.2	Standard expanding window functions	2260
34.16.2.1	pandas.core.window.Expanding.count	2260
34.16.2.2	pandas.core.window.Expanding.sum	2261
34.16.2.3	pandas.core.window.Expanding.mean	2262
34.16.2.4	pandas.core.window.Expanding.median	2263
34.16.2.5	pandas.core.window.Expanding.var	2264
34.16.2.6	pandas.core.window.Expanding.std	2265
34.16.2.7	pandas.core.window.Expanding.min	2266
34.16.2.8	pandas.core.window.Expanding.max	2267
34.16.2.9	pandas.core.window.Expanding.corr	2267
34.16.2.10	pandas.core.window.Expanding.cov	2267

34.16.2.1	pandas.core.window.Expanding.skew	2268
34.16.2.2	pandas.core.window.Expanding.kurt	2268
34.16.2.3	pandas.core.window.Expanding.apply	2269
34.16.2.4	pandas.core.window.Expanding.aggregate	2269
34.16.2.5	pandas.core.window.Expanding.quantile	2271
34.16.3	Exponentially-weighted moving window functions	2272
34.16.3.1	pandas.core.window.EWM.mean	2272
34.16.3.2	pandas.core.window.EWM.std	2272
34.16.3.3	pandas.core.window.EWM.var	2272
34.16.3.4	pandas.core.window.EWM.corr	2273
34.16.3.5	pandas.core.window.EWM.cov	2273
34.17	GroupBy	2273
34.17.1	Indexing, iteration	2274
34.17.1.1	pandas.core.groupby.GroupBy.__iter__	2274
34.17.1.2	pandas.core.groupby.GroupBy.groups	2274
34.17.1.3	pandas.core.groupby.GroupBy.indices	2274
34.17.1.4	pandas.core.groupby.GroupBy.get_group	2274
34.17.1.5	pandas.Grouper	2274
34.17.2	Function application	2276
34.17.2.1	pandas.core.groupby.GroupBy.apply	2276
34.17.2.2	pandas.core.groupby.GroupBy.aggregate	2277
34.17.2.3	pandas.core.groupby.GroupBy.transform	2277
34.17.2.4	pandas.core.groupby.GroupBy.pipe	2277
34.17.3	Computations / Descriptive Stats	2279
34.17.3.1	pandas.core.groupby.GroupBy.all	2279
34.17.3.2	pandas.core.groupby.GroupBy.any	2279
34.17.3.3	pandas.core.groupby.GroupBy.bfill	2280
34.17.3.4	pandas.core.groupby.GroupBy.count	2280
34.17.3.5	pandas.core.groupby.GroupBy.cumcount	2280
34.17.3.6	pandas.core.groupby.GroupBy.ffill	2281
34.17.3.7	pandas.core.groupby.GroupBy.first	2281
34.17.3.8	pandas.core.groupby.GroupBy.head	2281
34.17.3.9	pandas.core.groupby.GroupBy.last	2282
34.17.3.10	pandas.core.groupby.GroupBy.max	2282
34.17.3.11	pandas.core.groupby.GroupBy.mean	2282
34.17.3.12	pandas.core.groupby.GroupBy.median	2282
34.17.3.13	pandas.core.groupby.GroupBy.min	2283
34.17.3.14	pandas.core.groupby.GroupBy.ngroup	2283
34.17.3.15	pandas.core.groupby.GroupBy.nth	2284
34.17.3.16	pandas.core.groupby.GroupBy.ohlc	2285
34.17.3.17	pandas.core.groupby.GroupBy.prod	2285
34.17.3.18	pandas.core.groupby.GroupBy.rank	2285
34.17.3.19	pandas.core.groupby.GroupBy.pct_change	2286
34.17.3.20	pandas.core.groupby.GroupBy.size	2286
34.17.3.21	pandas.core.groupby.GroupBy.sem	2286
34.17.3.22	pandas.core.groupby.GroupBy.std	2287
34.17.3.23	pandas.core.groupby.GroupBy.sum	2287
34.17.3.24	pandas.core.groupby.GroupBy.var	2287
34.17.3.25	pandas.core.groupby.GroupBy.tail	2287
34.17.3.26	pandas.core.groupby.DataFrameGroupBy.agg	2289
34.17.3.27	pandas.core.groupby.DataFrameGroupBy.all	2290
34.17.3.28	pandas.core.groupby.DataFrameGroupBy.any	2291
34.17.3.29	pandas.core.groupby.DataFrameGroupBy.bfill	2291
34.17.3.30	pandas.core.groupby.DataFrameGroupBy.corr	2291



34.17.3.31	pandas.core.groupby.DataFrameGroupBy.count	2291
34.17.3.32	pandas.core.groupby.DataFrameGroupBy.cov	2291
34.17.3.33	pandas.core.groupby.DataFrameGroupBy.cummax	2293
34.17.3.34	pandas.core.groupby.DataFrameGroupBy.cummin	2293
34.17.3.35	pandas.core.groupby.DataFrameGroupBy.cumprod	2293
34.17.3.36	pandas.core.groupby.DataFrameGroupBy.cumsum	2293
34.17.3.37	pandas.core.groupby.DataFrameGroupBy.describe	2294
34.17.3.38	pandas.core.groupby.DataFrameGroupBy.diff	2297
34.17.3.39	pandas.core.groupby.DataFrameGroupBy.ffill	2299
34.17.3.40	pandas.core.groupby.DataFrameGroupBy.fillna	2299
34.17.3.41	pandas.core.groupby.DataFrameGroupBy.filter	2301
34.17.3.42	pandas.core.groupby.DataFrameGroupBy.hist	2301
34.17.3.43	pandas.core.groupby.DataFrameGroupBy.idxmax	2303
34.17.3.44	pandas.core.groupby.DataFrameGroupBy.idxmin	2303
34.17.3.45	pandas.core.groupby.DataFrameGroupBy.mad	2304
34.17.3.46	pandas.core.groupby.DataFrameGroupBy.pct_change	2304
34.17.3.47	pandas.core.groupby.DataFrameGroupBy.plot	2304
34.17.3.48	pandas.core.groupby.DataFrameGroupBy.quantile	2304
34.17.3.49	pandas.core.groupby.DataFrameGroupBy.rank	2306
34.17.3.50	pandas.core.groupby.DataFrameGroupBy.resample	2306
34.17.3.51	pandas.core.groupby.DataFrameGroupBy.shift	2307
34.17.3.52	pandas.core.groupby.DataFrameGroupBy.size	2307
34.17.3.53	pandas.core.groupby.DataFrameGroupBy.skew	2307
34.17.3.54	pandas.core.groupby.DataFrameGroupBy.take	2307
34.17.3.55	pandas.core.groupby.DataFrameGroupBy.tshift	2309
34.17.3.56	pandas.core.groupby.SeriesGroupBy.nlargest	2310
34.17.3.57	pandas.core.groupby.SeriesGroupBy.nsmallest	2310
34.17.3.58	pandas.core.groupby.SeriesGroupBy.nunique	2311
34.17.3.59	pandas.core.groupby.SeriesGroupBy.unique	2311
34.17.3.60	pandas.core.groupby.SeriesGroupBy.value_counts	2312
34.17.3.61	pandas.core.groupby.SeriesGroupBy.is_monotonic_increasing	2312
34.17.3.62	pandas.core.groupby.SeriesGroupBy.is_monotonic_decreasing	2312
34.17.3.63	pandas.core.groupby.DataFrameGroupBy.corrwith	2313
34.17.3.64	pandas.core.groupby.DataFrameGroupBy.boxplot	2313
34.18	Resampling	2314
34.18.1	Indexing, iteration	2314
34.18.1.1	pandas.core.resample.Resampler.__iter__	2314
34.18.1.2	pandas.core.resample.Resampler.groups	2314
34.18.1.3	pandas.core.resample.Resampler.indices	2315
34.18.1.4	pandas.core.resample.Resampler.get_group	2315
34.18.2	Function application	2315
34.18.2.1	pandas.core.resample.Resampler.apply	2315
34.18.2.2	pandas.core.resample.Resampler.aggregate	2317
34.18.2.3	pandas.core.resample.Resampler.transform	2318
34.18.2.4	pandas.core.resample.Resampler.pipe	2318
34.18.3	Upsampling	2319
34.18.3.1	pandas.core.resample.Resampler.ffill	2320
34.18.3.2	pandas.core.resample.Resampler.backfill	2320
34.18.3.3	pandas.core.resample.Resampler.bfill	2322
34.18.3.4	pandas.core.resample.Resampler.pad	2324
34.18.3.5	pandas.core.resample.Resampler.nearest	2324
34.18.3.6	pandas.core.resample.Resampler.fillna	2324
34.18.3.7	pandas.core.resample.Resampler.asfreq	2327
34.18.3.8	pandas.core.resample.Resampler.interpolate	2327

34.18.4	Computations / Descriptive Stats . . . . .	2329
34.18.4.1	pandas.core.resample.Resampler.count . . . . .	2329
34.18.4.2	pandas.core.resample.Resampler.nunique . . . . .	2329
34.18.4.3	pandas.core.resample.Resampler.first . . . . .	2329
34.18.4.4	pandas.core.resample.Resampler.last . . . . .	2330
34.18.4.5	pandas.core.resample.Resampler.max . . . . .	2330
34.18.4.6	pandas.core.resample.Resampler.mean . . . . .	2330
34.18.4.7	pandas.core.resample.Resampler.median . . . . .	2330
34.18.4.8	pandas.core.resample.Resampler.min . . . . .	2330
34.18.4.9	pandas.core.resample.Resampler.ohlc . . . . .	2330
34.18.4.10	pandas.core.resample.Resampler.prod . . . . .	2331
34.18.4.11	pandas.core.resample.Resampler.size . . . . .	2331
34.18.4.12	pandas.core.resample.Resampler.sem . . . . .	2331
34.18.4.13	pandas.core.resample.Resampler.std . . . . .	2331
34.18.4.14	pandas.core.resample.Resampler.sum . . . . .	2331
34.18.4.15	pandas.core.resample.Resampler.var . . . . .	2332
34.19	Style . . . . .	2332
34.19.1	Styler Constructor . . . . .	2332
34.19.1.1	pandas.io.formats.style.Styler . . . . .	2332
34.19.2	Styler Attributes . . . . .	2344
34.19.2.1	pandas.io.formats.style.Styler.env . . . . .	2344
34.19.2.2	pandas.io.formats.style.Styler.template . . . . .	2344
34.19.2.3	pandas.io.formats.style.Styler.loader . . . . .	2344
34.19.3	Style Application . . . . .	2344
34.19.4	Builtin Styles . . . . .	2345
34.19.5	Style Export and Import . . . . .	2345
34.20	Plotting . . . . .	2345
34.20.1	pandas.plotting.andrews_curves . . . . .	2345
34.20.2	pandas.plotting.bootstrap_plot . . . . .	2346
34.20.3	pandas.plotting.deregister_matplotlib_converters . . . . .	2347
34.20.4	pandas.plotting.lag_plot . . . . .	2347
34.20.5	pandas.plotting.parallel_coordinates . . . . .	2347
34.20.6	pandas.plotting.radviz . . . . .	2348
34.20.7	pandas.plotting.register_matplotlib_converters . . . . .	2349
34.20.8	pandas.plotting.scatter_matrix . . . . .	2350
34.21	General utility functions . . . . .	2351
34.21.1	Working with options . . . . .	2351
34.21.1.1	pandas.describe_option . . . . .	2351
34.21.1.2	pandas.reset_option . . . . .	2355
34.21.1.3	pandas.get_option . . . . .	2358
34.21.1.4	pandas.set_option . . . . .	2362
34.21.1.5	pandas.option_context . . . . .	2366
34.21.2	Testing functions . . . . .	2366
34.21.2.1	pandas.testing.assert_frame_equal . . . . .	2366
34.21.2.2	pandas.testing.assert_series_equal . . . . .	2367
34.21.2.3	pandas.testing.assert_index_equal . . . . .	2368
34.21.3	Exceptions and warnings . . . . .	2368
34.21.3.1	pandas.errors.DtypeWarning . . . . .	2369
34.21.3.2	pandas.errors.EmptyDataError . . . . .	2370
34.21.3.3	pandas.errors.OutOfBoundsDatetime . . . . .	2370
34.21.3.4	pandas.errors.ParserError . . . . .	2370
34.21.3.5	pandas.errors.ParserWarning . . . . .	2370
34.21.3.6	pandas.errors.PerformanceWarning . . . . .	2371
34.21.3.7	pandas.errors.UnsortedIndexError . . . . .	2371



34.21.3.8	pandas.errors.UnsupportedFunctionCall	2371
34.21.4	Data types related functionality	2371
34.21.4.1	pandas.api.types.union_categoricals	2372
34.21.4.2	pandas.api.types.infer_dtype	2373
34.21.4.3	pandas.api.types.pandas_dtype	2376
34.21.4.4	pandas.api.types.is_bool_dtype	2377
34.21.4.5	pandas.api.types.is_categorical_dtype	2377
34.21.4.6	pandas.api.types.is_complex_dtype	2378
34.21.4.7	pandas.api.types.is_datetime64_any_dtype	2378
34.21.4.8	pandas.api.types.is_datetime64_dtype	2379
34.21.4.9	pandas.api.types.is_datetime64_ns_dtype	2379
34.21.4.10	pandas.api.types.is_datetime64tz_dtype	2380
34.21.4.11	pandas.api.types.is_extension_type	2381
34.21.4.12	pandas.api.types.is_float_dtype	2381
34.21.4.13	pandas.api.types.is_int64_dtype	2382
34.21.4.14	pandas.api.types.is_integer_dtype	2383
34.21.4.15	pandas.api.types.is_interval_dtype	2383
34.21.4.16	pandas.api.types.is_numeric_dtype	2384
34.21.4.17	pandas.api.types.is_object_dtype	2384
34.21.4.18	pandas.api.types.is_period_dtype	2385
34.21.4.19	pandas.api.types.is_signed_integer_dtype	2385
34.21.4.20	pandas.api.types.is_string_dtype	2386
34.21.4.21	pandas.api.types.is_timedelta64_dtype	2387
34.21.4.22	pandas.api.types.is_timedelta64_ns_dtype	2387
34.21.4.23	pandas.api.types.is_unsigned_integer_dtype	2388
34.21.4.24	pandas.api.types.is_sparse	2388
34.21.4.25	pandas.api.types.is_dict_like	2389
34.21.4.26	pandas.api.types.is_file_like	2389
34.21.4.27	pandas.api.types.is_list_like	2390
34.21.4.28	pandas.api.types.is_named_tuple	2390
34.21.4.29	pandas.api.types.is_iterator	2391
34.21.4.30	pandas.api.types.is_bool	2392
34.21.4.31	pandas.api.types.is_categorical	2392
34.21.4.32	pandas.api.types.is_complex	2392
34.21.4.33	pandas.api.types.is_datetimetz	2392
34.21.4.34	pandas.api.types.is_float	2393
34.21.4.35	pandas.api.types.is_hashable	2393
34.21.4.36	pandas.api.types.is_integer	2393
34.21.4.37	pandas.api.types.is_interval	2393
34.21.4.38	pandas.api.types.is_number	2393
34.21.4.39	pandas.api.types.is_period	2394
34.21.4.40	pandas.api.types.is_re	2394
34.21.4.41	pandas.api.types.is_re_compilable	2395
34.21.4.42	pandas.api.types.is_scalar	2395
34.22	Extensions	2395
34.22.1	pandas.api.extensions.register_dataframe_accessor	2396
34.22.2	pandas.api.extensions.register_series_accessor	2397
34.22.3	pandas.api.extensions.register_index_accessor	2398
34.22.4	pandas.api.extensions.ExtensionDtype	2399
34.22.4.1	pandas.api.extensions.ExtensionDtype.kind	2400
34.22.4.2	pandas.api.extensions.ExtensionDtype.name	2400
34.22.4.3	pandas.api.extensions.ExtensionDtype.names	2400
34.22.4.4	pandas.api.extensions.ExtensionDtype.type	2400
34.22.4.5	pandas.api.extensions.ExtensionDtype.construct_from_string	2400

34.22.4.6	pandas.api.extensions.ExtensionDtype.is_dtype	2401
34.22.5	pandas.api.extensions.ExtensionArray	2401
34.22.5.1	pandas.api.extensions.ExtensionArray.dtype	2403
34.22.5.2	pandas.api.extensions.ExtensionArray.nbytes	2403
34.22.5.3	pandas.api.extensions.ExtensionArray.ndim	2403
34.22.5.4	pandas.api.extensions.ExtensionArray.shape	2403
34.22.5.5	pandas.api.extensions.ExtensionArray.argsort	2403
34.22.5.6	pandas.api.extensions.ExtensionArray.astype	2404
34.22.5.7	pandas.api.extensions.ExtensionArray.copy	2404
34.22.5.8	pandas.api.extensions.ExtensionArray.factorize	2404
34.22.5.9	pandas.api.extensions.ExtensionArray.fillna	2405
34.22.5.10	pandas.api.extensions.ExtensionArray.isna	2405
34.22.5.11	pandas.api.extensions.ExtensionArray.take	2405
34.22.5.12	pandas.api.extensions.ExtensionArray.unique	2407
34.22.6	pandas.Index.asi8	2407
34.22.7	pandas.Index.holds_integer	2407
34.22.8	pandas.Index.is_type_compatible	2407
34.22.9	pandas.Index.nlevels	2407
34.22.10	pandas.Index.sort	2407
34.22.11	pandas.Panel.agg	2407
34.22.12	pandas.Panel.aggregate	2407
34.22.13	pandas.Panel.is_copy	2407
34.22.14	pandas.Series.imag	2408
34.22.15	pandas.Series.real	2408
<b>35</b>	<b>Developer</b>	<b>2409</b>
35.1	Storing pandas DataFrame objects in Apache Parquet format	2409
<b>36</b>	<b>Internals</b>	<b>2413</b>
36.1	Indexing	2413
36.1.1	MultiIndex	2414
36.1.2	Values	2414
36.2	Subclassing pandas Data Structures	2415
<b>37</b>	<b>Extending Pandas</b>	<b>2417</b>
37.1	Registering Custom Accessors	2417
37.2	Extension Types	2418
37.2.1	ExtensionDtype	2418
37.2.2	ExtensionArray	2418
37.3	Subclassing pandas Data Structures	2419
37.3.1	Override Constructor Properties	2419
37.3.2	Define Original Properties	2421
<b>38</b>	<b>Release Notes</b>	<b>2423</b>
38.1	pandas 0.23.2	2423
38.1.1	Thanks	2423
38.2	pandas 0.23.1	2424
38.2.1	Thanks	2424
38.3	pandas 0.23.0	2425
38.3.1	Thanks	2425
38.4	pandas 0.22.0	2435
38.5	pandas 0.21.1	2435
38.5.1	Thanks	2435
38.5.1.1	Contributors	2435
38.6	pandas 0.21.0	2437

38.6.1	Thanks	2437
38.6.1.1	Contributors	2437
38.7	pandas 0.20.0 / 0.20.1	2443
38.7.1	Thanks	2444
38.8	pandas 0.19.2	2449
38.8.1	Thanks	2450
38.9	pandas 0.19.1	2451
38.9.1	Thanks	2451
38.10	pandas 0.19.0	2452
38.10.1	Thanks	2452
38.11	pandas 0.18.1	2456
38.11.1	Thanks	2456
38.12	pandas 0.18.0	2458
38.12.1	Thanks	2458
38.13	pandas 0.17.1	2461
38.13.1	Thanks	2461
38.14	pandas 0.17.0	2463
38.14.1	Thanks	2464
38.15	pandas 0.16.2	2467
38.15.1	Thanks	2467
38.16	pandas 0.16.1	2468
38.16.1	Thanks	2468
38.17	pandas 0.16.0	2470
38.17.1	Thanks	2470
38.18	pandas 0.15.2	2472
38.18.1	Thanks	2472
38.19	pandas 0.15.1	2474
38.19.1	Thanks	2474
38.20	pandas 0.15.0	2475
38.20.1	Thanks	2475
38.21	pandas 0.14.1	2477
38.21.1	Thanks	2478
38.22	pandas 0.14.0	2479
38.22.1	Thanks	2480
38.23	pandas 0.13.1	2482
38.23.1	New Features	2482
38.23.2	API Changes	2482
38.23.3	Experimental Features	2483
38.23.4	Improvements to existing features	2483
38.23.5	Bug Fixes	2484
38.24	pandas 0.13.0	2485
38.24.1	New Features	2485
38.24.2	Experimental Features	2486
38.24.3	Improvements to existing features	2486
38.24.4	API Changes	2489
38.24.5	Internal Refactoring	2492
38.24.6	Bug Fixes	2494
38.25	pandas 0.12.0	2499
38.25.1	New Features	2499
38.25.2	Improvements to existing features	2500
38.25.3	API Changes	2501
38.25.4	Experimental Features	2502
38.25.5	Bug Fixes	2503
38.26	pandas 0.11.0	2506

38.26.1	New Features	2506
38.26.2	Improvements to existing features	2506
38.26.3	API Changes	2508
38.26.4	Bug Fixes	2509
38.27	pandas 0.10.1	2512
38.27.1	New Features	2512
38.27.2	API Changes	2512
38.27.3	Improvements to existing features	2512
38.27.4	Bug Fixes	2513
38.28	pandas 0.10.0	2514
38.28.1	New Features	2514
38.28.2	Experimental Features	2515
38.28.3	API Changes	2515
38.28.4	Improvements to existing features	2516
38.28.5	Bug Fixes	2517
38.29	pandas 0.9.1	2519
38.29.1	New Features	2519
38.29.2	API Changes	2519
38.29.3	Improvements to existing features	2519
38.29.4	Bug Fixes	2520
38.30	pandas 0.9.0	2521
38.30.1	New Features	2521
38.30.2	Improvements to existing features	2522
38.30.3	API Changes	2522
38.30.4	Bug Fixes	2523
38.31	pandas 0.8.1	2526
38.31.1	New Features	2526
38.31.2	Improvements to existing features	2527
38.31.3	Bug Fixes	2527
38.32	pandas 0.8.0	2528
38.32.1	New Features	2528
38.32.2	Improvements to existing features	2530
38.32.3	API Changes	2531
38.32.4	Bug Fixes	2531
38.33	pandas 0.7.3	2533
38.33.1	New Features	2533
38.33.2	API Changes	2533
38.33.3	Bug Fixes	2534
38.34	pandas 0.7.2	2534
38.34.1	New Features	2534
38.34.2	API Changes	2534
38.34.3	Improvements to existing features	2535
38.34.4	Bug Fixes	2535
38.35	pandas 0.7.1	2536
38.35.1	New Features	2536
38.35.2	Improvements to existing features	2536
38.35.3	Bug Fixes	2536
38.36	pandas 0.7.0	2537
38.36.1	New Features	2537
38.36.2	API Changes	2538
38.36.3	Improvements to existing features	2539
38.36.4	Bug Fixes	2540
38.36.5	Thanks	2542
38.37	pandas 0.6.1	2543

38.37.1	API Changes	2543
38.37.2	New Features	2543
38.37.3	Improvements to existing features	2544
38.37.4	Bug Fixes	2544
38.37.5	Thanks	2545
38.38	pandas 0.6.0	2545
38.38.1	API Changes	2546
38.38.2	New Features	2546
38.38.3	Improvements to existing features	2547
38.38.4	Bug Fixes	2547
38.38.5	Thanks	2549
38.39	pandas 0.5.0	2549
38.39.1	API Changes	2549
38.39.2	Deprecations Removed	2551
38.39.3	New Features	2551
38.39.4	Improvements to existing features	2551
38.39.5	Bug Fixes	2552
38.39.6	Thanks	2553
38.40	pandas 0.4.3	2553
38.40.1	New Features	2553
38.40.2	Improvements to existing features	2553
38.40.3	API Changes	2554
38.40.4	Bug Fixes	2554
38.40.5	Thanks	2554
38.41	pandas 0.4.2	2554
38.41.1	New Features	2554
38.41.2	Improvements to existing features	2555
38.41.3	API Changes	2555
38.41.4	Bug Fixes	2555
38.41.5	Thanks	2555
38.42	pandas 0.4.1	2555
38.42.1	New Features	2556
38.42.2	Improvements to existing features	2556
38.42.3	API Changes	2556
38.42.4	Bug Fixes	2556
38.42.5	Thanks	2557
38.43	pandas 0.4.0	2557
38.43.1	New Features	2557
38.43.2	Improvements to existing features	2558
38.43.3	API Changes	2559
38.43.4	Bug Fixes	2560
38.43.5	Thanks	2561
38.44	pandas 0.3.0	2562
38.44.1	New features	2562
38.44.2	Improvements to existing features	2562
38.44.3	API Changes	2562
38.44.4	Bug Fixes	2563

<b>Bibliography</b>	<b>2565</b>
---------------------	-------------

<b>Python Module Index</b>	<b>2567</b>
----------------------------	-------------



PDF Version

Zipped HTML

Date: Jul 05, 2018 Version: 0.23.2

Binary Installers: <https://pypi.org/project/pandas>

Source Repository: <http://github.com/pandas-dev/pandas>

Issues & Ideas: <https://github.com/pandas-dev/pandas/issues>

Q&A Support: <http://stackoverflow.com/questions/tagged/pandas>

Developer Mailing List: <http://groups.google.com/group/pydata>

**pandas** is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, **real world** data analysis in Python. Additionally, it has the broader goal of becoming **the most powerful and flexible open source data analysis / manipulation tool available in any language**. It is already well on its way toward this goal.

pandas is well suited for many different kinds of data:

- Tabular data with heterogeneously-typed columns, as in an SQL table or Excel spreadsheet
- Ordered and unordered (not necessarily fixed-frequency) time series data.
- Arbitrary matrix data (homogeneously typed or heterogeneous) with row and column labels
- Any other form of observational / statistical data sets. The data actually need not be labeled at all to be placed into a pandas data structure

The two primary data structures of pandas, *Series* (1-dimensional) and *DataFrame* (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering. For R users, *DataFrame* provides everything that R’s `data.frame` provides and much more. pandas is built on top of NumPy and is intended to integrate well within a scientific computing environment with many other 3rd party libraries.

Here are just a few of the things that pandas does well:

- Easy handling of **missing data** (represented as NaN) in floating point as well as non-floating point data
- Size mutability: columns can be **inserted and deleted** from DataFrame and higher dimensional objects
- Automatic and explicit **data alignment**: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let *Series*, *DataFrame*, etc. automatically align the data for you in computations
- Powerful, flexible **group by** functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
- Make it **easy to convert** ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects
- Intelligent label-based **slicing, fancy indexing**, and **subsetting** of large data sets
- Intuitive **merging** and **joining** data sets
- Flexible **reshaping** and pivoting of data sets
- **Hierarchical** labeling of axes (possible to have multiple labels per tick)
- Robust IO tools for loading data from **flat files** (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast **HDF5 format**
- **Time series**-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

Many of these principles are here to address the shortcomings frequently experienced using other languages / scientific research environments. For data scientists, working with data is typically divided into multiple stages: munging and cleaning data, analyzing / modeling it, then organizing the results of the analysis into a form suitable for plotting or tabular display. pandas is the ideal tool for all of these tasks.

Some other notes

- pandas is **fast**. Many of the low-level algorithmic bits have been extensively tweaked in [Cython](#) code. However, as with anything else generalization usually sacrifices performance. So if you focus on one feature for your application you may be able to create a faster specialized tool.
- pandas is a dependency of [statsmodels](#), making it an important part of the statistical computing ecosystem in Python.
- pandas has been used extensively in production in financial applications.

---

**Note:** This documentation assumes general familiarity with NumPy. If you haven't used NumPy much or at all, do invest some time in [learning about NumPy](#) first.

---

See the package overview for more detail about what's in the library.



## WHAT'S NEW

These are new features and improvements of note in each release.

### 1.1 v0.23.2

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

---

**Note:** Pandas 0.23.2 is first pandas release that's compatible with Python 3.7 ([GH20552](#))

---

#### What's new in v0.23.2

- *Logical Reductions over Entire DataFrame*
- *Fixed Regressions*
- *Build Changes*
- *Bug Fixes*

#### 1.1.1 Logical Reductions over Entire DataFrame

`DataFrame.all()` and `DataFrame.any()` now accept `axis=None` to reduce over all axes to a scalar ([GH19976](#))

```
In [1]: df = pd.DataFrame({"A": [1, 2], "B": [True, False]})
In [2]: df.all(axis=None)
Out[2]: False
```

This also provides compatibility with NumPy 1.15, which now dispatches to `DataFrame.all`. With NumPy 1.15 and pandas 0.23.1 or earlier, `numpy.all()` will no longer reduce over every axis:

```
>>> # NumPy 1.15, pandas 0.23.1
>>> np.any(pd.DataFrame({"A": [False], "B": [False]}))
A    False
B    False
dtype: bool
```

With pandas 0.23.2, that will correctly return False, as it did with NumPy < 1.15.

```
In [3]: np.any(pd.DataFrame({"A": [False], "B": [False]}))
Out[3]: False
```

### 1.1.2 Fixed Regressions

- Fixed regression in `to_csv()` when handling file-like object incorrectly (GH21471)
- Re-allowed duplicate level names of a `MultiIndex`. Accessing a level that has a duplicate name by name still raises an error (GH19029).
- Bug in both `DataFrame.first_valid_index()` and `Series.first_valid_index()` raised for a row index having duplicate values (GH21441)
- Fixed printing of DataFrames with hierarchical columns with long names (GH21180)
- Fixed regression in `reindex()` and `groupby()` with a `MultiIndex` or multiple keys that contains categorical datetime-like values (GH21390).
- Fixed regression in unary negative operations with object dtype (GH21380)
- Bug in `Timestamp.ceil()` and `Timestamp.floor()` when timestamp is a multiple of the rounding frequency (GH21262)
- Fixed regression in `to_clipboard()` that defaulted to copying dataframes with space delimited instead of tab delimited (GH21104)

### 1.1.3 Build Changes

- The source and binary distributions no longer include test data files, resulting in smaller download sizes. Tests relying on these data files will be skipped when using `pandas.test()`. (GH19320)

### 1.1.4 Bug Fixes

#### Conversion

- Bug in constructing `Index` with an iterator or generator (GH21470)
- Bug in `Series.nlargest()` for signed and unsigned integer dtypes when the minimum value is present (GH21426)

#### Indexing

- Bug in `Index.get_indexer_non_unique()` with categorical key (GH21448)
- Bug in comparison operations for `MultiIndex` where error was raised on equality / inequality comparison involving a `MultiIndex` with `nlevels == 1` (GH21149)
- Bug in `DataFrame.drop()` behaviour is not consistent for unique and non-unique indexes (GH21494)
- Bug in `DataFrame.duplicated()` with a large number of columns causing a ‘maximum recursion depth exceeded’ (GH21524).

#### I/O

- Bug in `read_csv()` that caused it to incorrectly raise an error when `nrows=0`, `low_memory=True`, and `index_col` was not None (GH21141)
- Bug in `json_normalize()` when formatting the `record_prefix` with integer columns (GH21536)

### Categorical

- Bug in rendering *Series* with Categorical dtype in rare conditions under Python 2.7 (GH21002)

### Timezones

- Bug in *Timestamp* and *DatetimeIndex* where passing a *Timestamp* localized after a DST transition would return a datetime before the DST transition (GH20854)
- Bug in comparing *DataFrame*'s with tz-aware `:class:`DatetimeIndex` columns with a DST transition that raised a *KeyError* (GH19970)

### Timedelta

- Bug in *Timedelta* where non-zero timedeltas shorter than 1 microsecond were considered False (GH21484)

## 1.2 v0.23.1

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

### What's new in v0.23.1

- *Fixed Regressions*
- *Performance Improvements*
- *Bug Fixes*

### 1.2.1 Fixed Regressions

#### Comparing Series with `datetime.date`

We've reverted a 0.23.0 change to comparing a *Series* holding datetimes and a `datetime.date` object (GH21152). In pandas 0.22 and earlier, comparing a *Series* holding datetimes and `datetime.date` objects would coerce the `datetime.date` to a `datetime` before comparing. This was inconsistent with Python, NumPy, and *DatetimeIndex*, which never consider a `datetime` and `datetime.date` equal.

In 0.23.0, we unified operations between *DatetimeIndex* and *Series*, and in the process changed comparisons between a *Series* of datetimes and `datetime.date` without warning.

We've temporarily restored the 0.22.0 behavior, so datetimes and dates may again compare equal, but restore the 0.23.0 behavior in a future release.

To summarize, here's the behavior in 0.22.0, 0.23.0, 0.23.1:

```
# 0.22.0... Silently coerce the datetime.date
>>> Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
0      True
1     False
dtype: bool

# 0.23.0... Do not coerce the datetime.date
>>> Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
0     False
1     False
```

(continues on next page)

(continued from previous page)

```

dtype: bool

# 0.23.1... Coerce the datetime.date with a warning
>>> Series(pd.date_range('2017', periods=2)) == datetime.date(2017, 1, 1)
/bin/python:1: FutureWarning: Comparing Series of datetimes with 'datetime.date'.
↳Currently, the
'datetime.date' is coerced to a datetime. In the future pandas will
not coerce, and the values not compare equal to the 'datetime.date'.
To retain the current behavior, convert the 'datetime.date' to a
datetime with 'pd.Timestamp'.
  #!/bin/python3
0      True
1     False
dtype: bool

```

In addition, ordering comparisons will raise a `TypeError` in the future.

### Other Fixes

- Reverted the ability of `to_sql()` to perform multivalue inserts as this caused regression in certain cases (GH21103). In the future this will be made configurable.
- Fixed regression in the `DatetimeIndex.date` and `DatetimeIndex.time` attributes in case of timezone-aware data: `DatetimeIndex.time` returned a tz-aware time instead of tz-naive (GH21267) and `DatetimeIndex.date` returned incorrect date when the input date has a non-UTC timezone (GH21230).
- Fixed regression in `pandas.io.json.json_normalize()` when called with `None` values in nested levels in JSON, and to not drop keys with value as `None` (GH21158, GH21356).
- Bug in `to_csv()` causes encoding error when compression and encoding are specified (GH21241, GH21118)
- Bug preventing pandas from being importable with -OO optimization (GH21071)
- Bug in `Categorical.fillna()` incorrectly raising a `TypeError` when `value` the individual categories are iterable and `value` is an iterable (GH21097, GH19788)
- Fixed regression in constructors coercing NA values like `None` to strings when passing `dtype=str` (GH21083)
- Regression in `pivot_table()` where an ordered `Categorical` with missing values for the pivot's index would give a mis-aligned result (GH21133)
- Fixed regression in merging on boolean index/columns (GH21119).

## 1.2.2 Performance Improvements

- Improved performance of `CategoricalIndex.is_monotonic_increasing()`, `CategoricalIndex.is_monotonic_decreasing()` and `CategoricalIndex.is_monotonic()` (GH21025)
- Improved performance of `CategoricalIndex.is_unique()` (GH21107)

## 1.2.3 Bug Fixes

### Groupby/Resample/Rolling

- Bug in `DataFrame.agg()` where applying multiple aggregation functions to a `DataFrame` with duplicated column names would cause a stack overflow (GH21063)

- Bug in `pandas.core.groupby.GroupBy.ffill()` and `pandas.core.groupby.GroupBy.bfill()` where the fill within a grouping would not always be applied as intended due to the implementations' use of a non-stable sort (GH21207)
- Bug in `pandas.core.groupby.GroupBy.rank()` where results did not scale to 100% when specifying `method='dense'` and `pct=True`
- Bug in `pandas.DataFrame.rolling()` and `pandas.Series.rolling()` which incorrectly accepted a 0 window size rather than raising (GH21286)

### Data-type specific

- Bug in `Series.str.replace()` where the method throws `TypeError` on Python 3.5.2 (GH21078)
- Bug in `Timedelta`: where passing a float with a unit would prematurely round the float precision (GH14156)
- Bug in `pandas.testing.assert_index_equal()` which raised `AssertionError` incorrectly, when comparing two `CategoricalIndex` objects with param `check_categorical=False` (GH19776)

### Sparse

- Bug in `SparseArray.shape` which previously only returned the shape `SparseArray.sp_values` (GH21126)

### Indexing

- Bug in `Series.reset_index()` where appropriate error was not raised with an invalid level name (GH20925)
- Bug in `interval_range()` when start/periods or end/periods are specified with float start or end (GH21161)
- Bug in `MultiIndex.set_names()` where error raised for a `MultiIndex` with `nlevels == 1` (GH21149)
- Bug in `IntervalIndex` constructors where creating an `IntervalIndex` from categorical data was not fully supported (GH21243, GH21253)
- Bug in `MultiIndex.sort_index()` which was not guaranteed to sort correctly with `level=1`; this was also causing data misalignment in particular `DataFrame.stack()` operations (GH20994, GH20945, GH21052)

### Plotting

- New keywords (`sharex`, `sharey`) to turn on/off sharing of x/y-axis by subplots generated with `pandas.DataFrame().groupby().boxplot()` (GH20968)

### I/O

- Bug in IO methods specifying `compression='zip'` which produced uncompressed zip archives (GH17778, GH21144)
- Bug in `DataFrame.to_stata()` which prevented exporting DataFrames to buffers and most file-like objects (GH21041)
- Bug in `read_stata()` and `StataReader` which did not correctly decode utf-8 strings on Python 3 from Stata 14 files (dta version 118) (GH21244)
- Bug in IO JSON `read_json()` reading empty JSON schema with `orient='table'` back to `DataFrame` caused an error (GH21287)

### Reshaping

- Bug in `concat()` where error was raised in concatenating *Series* with numpy scalar and tuple names (GH21015)
- Bug in `concat()` warning message providing the wrong guidance for future behavior (GH21101)

#### Other

- Tab completion on *Index* in IPython no longer outputs deprecation warnings (GH21125)
- Bug preventing pandas being used on Windows without C++ redistributable installed (GH21106)

## 1.3 v0.23.0 (May 15, 2018)

This is a major release from 0.22.0 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- *Round-trippable JSON format with 'table' orient.*
- *Instantiation from dicts respects order for Python 3.6+.*
- *Dependent column arguments for assign.*
- *Merging / sorting on a combination of columns and index levels.*
- *Extending Pandas with custom types.*
- *Excluding unobserved categories from groupby.*
- *Changes to make output shape of DataFrame.apply consistent.*

Check the *API Changes* and *deprecations* before updating.

**Warning:** Starting January 1, 2019, pandas feature releases will support Python 3 only. See *Plan for dropping Python 2.7* for more.

#### What's new in v0.23.0

- *New features*
  - *JSON read/write round-trippable with `orient='table'`*
  - *`.assign()` accepts dependent arguments*
  - *Merging on a combination of columns and index levels*
  - *Sorting by a combination of columns and index levels*
  - *Extending Pandas with Custom Types (Experimental)*
  - *New `observed` keyword for excluding unobserved categories in `groupby`*
  - *`Rolling/Expanding.apply()` accepts `raw=False` to pass a *Series* to the function*
  - *`DataFrame.interpolate` has gained the `limit_area` kwarg*
  - *`get_dummies` now supports `dtype` argument*
  - *`Timedelta` mod method*

- `.rank()` handles `inf` values when `NaN` are present
- `Series.str.cat` has gained the `join` kwarg
- `DataFrame.astype` performs column-wise conversion to `Categorical`
- Other Enhancements
- Backwards incompatible API changes
  - Dependencies have increased minimum versions
  - Instantiation from dicts preserves dict insertion order for python 3.6+
  - Deprecate `Panel`
  - `pandas.core.common` removals
  - Changes to make output of `DataFrame.apply` consistent
  - Concatenation will no longer sort
  - Build Changes
  - Index Division By Zero Fills Correctly
  - Extraction of matching patterns from strings
  - Default value for the `ordered` parameter of `CategoricalDtype`
  - Better pretty-printing of `DataFrames` in a terminal
  - Datetimelike API Changes
  - Other API Changes
- Deprecations
- Removal of prior version deprecations/changes
- Performance Improvements
- Documentation Changes
- Bug Fixes
  - `Categorical`
  - `Datetimelike`
  - `Timedelta`
  - `Timezones`
  - `Offsets`
  - `Numeric`
  - `Strings`
  - `Indexing`
  - `MultiIndex`
  - `I/O`
  - `Plotting`
  - `Groupby/Resample/Rolling`

- *Sparse*
- *Reshaping*
- *Other*

## 1.3.1 New features

### 1.3.1.1 JSON read/write round-trippable with `orient='table'`

A `DataFrame` can now be written to and subsequently read back via JSON while preserving metadata through usage of the `orient='table'` argument (see [GH18912](#) and [GH9146](#)). Previously, none of the available `orient` values guaranteed the preservation of dtypes and index names, amongst other metadata.

```
In [1]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
...:                      'bar': ['a', 'b', 'c', 'd'],
...:                      'baz': pd.date_range('2018-01-01', freq='d', periods=4),
...:                      'qux': pd.Categorical(['a', 'b', 'c', 'c'])
...:                      }, index=pd.Index(range(4), name='idx'))

In [2]: df
Out[2]:
```

	foo	bar	baz	qux
idx				
0	1	a	2018-01-01	a
1	2	b	2018-01-02	b
2	3	c	2018-01-03	c
3	4	d	2018-01-04	c

```
In [3]: df.dtypes
//////////
foo          int64
bar          object
baz    datetime64[ns]
qux          category
dtype: object

In [4]: df.to_json('test.json', orient='table')

In [5]: new_df = pd.read_json('test.json', orient='table')

In [6]: new_df
Out[6]:
```

	foo	bar	baz	qux
idx				
0	1	a	2018-01-01	a
1	2	b	2018-01-02	b
2	3	c	2018-01-03	c
3	4	d	2018-01-04	c

```
In [7]: new_df.dtypes
//////////
foo          int64
```

(continues on next page)





**Warning:** This may subtly change the behavior of your code when you're using `.assign()` to update an existing column. Previously, callables referring to other variables being updated would get the “old” values

Previous Behavior:

```
In [2]: df = pd.DataFrame({"A": [1, 2, 3]})

In [3]: df.assign(A=lambda df: df.A + 1, C=lambda df: df.A * -1)
Out[3]:
```

	A	C
0	2	-1
1	3	-2
2	4	-3

New Behavior:

```
In [16]: df.assign(A=df.A+1, C= lambda df: df.A* -1)
Out[16]:
```

	A	C
0	2	-2
1	3	-3
2	4	-4

### 1.3.1.3 Merging on a combination of columns and index levels

Strings passed to `DataFrame.merge()` as the `on`, `left_on`, and `right_on` parameters may now refer to either column names or index level names. This enables merging `DataFrame` instances on a combination of index levels and columns without resetting indexes. See the *Merge on columns and levels* documentation section. (GH14355)

```
In [17]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')

In [18]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
.....:                      'B': ['B0', 'B1', 'B2', 'B3'],
.....:                      'key2': ['K0', 'K1', 'K0', 'K1']},
.....:                      index=left_index)
.....:

In [19]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')

In [20]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
.....:                        'D': ['D0', 'D1', 'D2', 'D3'],
.....:                        'key2': ['K0', 'K0', 'K0', 'K1']},
.....:                        index=right_index)
.....:

In [21]: left.merge(right, on=['key1', 'key2'])
Out[21]:
```

	A	B	key2	C	D
key1					
K0	A0	B0	K0	C0	D0
K1	A2	B2	K0	C1	D1
K2	A3	B3	K1	C3	D3

#### 1.3.1.4 Sorting by a combination of columns and index levels

Strings passed to `DataFrame.sort_values()` as the `by` parameter may now refer to either column names or index level names. This enables sorting `DataFrame` instances by a combination of index levels and columns without resetting indexes. See the [Sorting by Indexes and Values](#) documentation section. (GH14353)

[illegible]

### 1.3.1.5 Extending Pandas with Custom Types (Experimental)

Pandas now supports storing array-like objects that aren't necessarily 1-D NumPy arrays as columns in a DataFrame or values in a Series. This allows third-party libraries to implement extensions to NumPy's types, similar to how pandas implemented categoricals, datetimes with timezones, periods, and intervals.

As a demonstration, we'll use `cyberpandas`, which provides an `IPArray` type for storing ip addresses.

```
In [1]: from cyberpandas import IPArray

In [2]: values = IPArray([
...:     0,
...:     3232235777,
...:     42540766452641154071740215577757643572
```

---

(continues on next page)

(continued from previous page)

```
...: ])
```

IPArray isn't a normal 1-D NumPy array, but because it's a pandas `~pandas.api.extension.ExtensionArray`, it can be stored properly inside pandas' containers.

```
In [3]: ser = pd.Series(values)

In [4]: ser
Out[4]:
0          0.0.0.0
1      192.168.1.1
2  2001:db8:85a3::8a2e:370:7334
dtype: ip
```

Notice that the dtype is `ip`. The missing value semantics of the underlying array are respected:

```
In [5]: ser.isna()
Out[5]:
0      True
1     False
2     False
dtype: bool
```

For more, see the [extension types](#) documentation. If you build an extension array, publicize it on our [ecosystem page](#).

### 1.3.1.6 New observed keyword for excluding unobserved categories in groupby

Grouping by a categorical includes the unobserved categories in the output. When grouping by multiple categorical columns, this means you get the cartesian product of all the categories, including combinations where there are no observations, which can result in a large number of groups. We have added a keyword `observed` to control this behavior, it defaults to `observed=False` for backward-compatibility. (GH14942, GH8138, GH15217, GH17594, GH8669, GH20583, GH20902)

```
In [27]: cat1 = pd.Categorical(["a", "a", "b", "b"],
...:                           categories=["a", "b", "z"], ordered=True)
...:
...:

In [28]: cat2 = pd.Categorical(["c", "d", "c", "d"],
...:                           categories=["c", "d", "y"], ordered=True)
...:
...:

In [29]: df = pd.DataFrame({"A": cat1, "B": cat2, "values": [1, 2, 3, 4]})

In [30]: df['C'] = ['foo', 'bar'] * 2

In [31]: df
Out[31]:
   A  B  values  C
0  a  c        1  foo
1  a  d        2  bar
2  b  c        3  foo
3  b  d        4  bar
```

To show all values, the previous behavior:

```
In [32]: df.groupby(['A', 'B', 'C'], observed=False).count()
```

```
Out [32]:
```

```
      values
A B C
a c bar    NaN
   foo    1.0
   d bar    1.0
   foo    NaN
   y bar    NaN
   foo    NaN
b c bar    NaN
...      ...
   y foo    NaN
z c bar    NaN
   foo    NaN
   d bar    NaN
   foo    NaN
   y bar    NaN
   foo    NaN
```

```
[18 rows x 1 columns]
```

To show only observed values:

```
In [33]: df.groupby(['A', 'B', 'C'], observed=True).count()
```

```
Out [33]:
```

```
      values
A B C
a c foo     1
   d bar     1
b c foo     1
   d bar     1
```

For pivoting operations, this behavior is *already* controlled by the `dropna` keyword:

```
In [34]: cat1 = pd.Categorical(["a", "a", "b", "b"],
.....:                        categories=["a", "b", "z"], ordered=True)
.....:
```

```
In [35]: cat2 = pd.Categorical(["c", "d", "c", "d"],
.....:                        categories=["c", "d", "y"], ordered=True)
.....:
```

```
In [36]: df = DataFrame({"A": cat1, "B": cat2, "values": [1, 2, 3, 4]})
```

```
In [37]: df
```

```
Out [37]:
```

```
   A B  values
0  a c       1
1  a d       2
2  b c       3
3  b d       4
```

```
In [38]: pd.pivot_table(df, values='values', index=['A', 'B'],
.....:                  dropna=True)
.....:
```

```
Out [38]:
```

(continues on next page)

(continued from previous page)

```

      values
A B
a c      1
  d      2
b c      3
  d      4

In [39]: pd.pivot_table(df, values='values', index=['A', 'B'],
      ....:               dropna=False)
      ....:

\\Out[39]:
↪
      values
A B
a c      1.0
  d      2.0
  y      NaN
b c      3.0
  d      4.0
  y      NaN
z c      NaN
  d      NaN
  y      NaN

```

### 1.3.1.7 Rolling/Expanding.apply() accepts raw=False to pass a Series to the function

`Series.rolling().apply()`, `DataFrame.rolling().apply()`, `Series.expanding().apply()`, and `DataFrame.expanding().apply()` have gained a `raw=None` parameter. This is similar to `DataFrame.apply()`. This parameter, if `True` allows one to send a `np.ndarray` to the applied function. If `False` a `Series` will be passed. The default is `None`, which preserves backward compatibility, so this will default to `True`, sending an `np.ndarray`. In a future version the default will be changed to `False`, sending a `Series`. (GH5071, GH20584)

```

In [40]: s = pd.Series(np.arange(5), np.arange(5) + 1)

In [41]: s
Out[41]:
1      0
2      1
3      2
4      3
5      4
dtype: int64

```

Pass a Series:

```

In [42]: s.rolling(2, min_periods=1).apply(lambda x: x.iloc[-1], raw=False)
Out[42]:
1      0.0
2      1.0
3      2.0
4      3.0
5      4.0
dtype: float64

```

Mimic the original behavior of passing a ndarray:

```
In [43]: s.rolling(2, min_periods=1).apply(lambda x: x[-1], raw=True)
Out[43]:
1    0.0
2    1.0
3    2.0
4    3.0
5    4.0
dtype: float64
```

### 1.3.1.8 DataFrame.interpolate has gained the limit\_area kwarg

`DataFrame.interpolate()` has gained a `limit_area` parameter to allow further control of which NaNs are replaced. Use `limit_area='inside'` to fill only NaNs surrounded by valid values or use `limit_area='outside'` to fill only NaNs outside the existing valid values while preserving those inside. (GH16284) See the [full documentation here](#).

```
In [44]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13, np.nan, np.
    ↪ nan])

In [45]: ser
Out[45]:
0    NaN
1    NaN
2    5.0
3    NaN
4    NaN
5    NaN
6   13.0
7    NaN
8    NaN
dtype: float64
```

Fill one consecutive inside value in both directions

```
In [46]: ser.interpolate(limit_direction='both', limit_area='inside', limit=1)
Out[46]:
0    NaN
1    NaN
2    5.0
3    7.0
4    NaN
5   11.0
6   13.0
7    NaN
8    NaN
dtype: float64
```

Fill all consecutive outside values backward

```
In [47]: ser.interpolate(limit_direction='backward', limit_area='outside')
Out[47]:
0    5.0
1    5.0
2    5.0
```

(continues on next page)

(continued from previous page)

```

3      NaN
4      NaN
5      NaN
6     13.0
7      NaN
8      NaN
dtype: float64

```

Fill all consecutive outside values in both directions

```

In [48]: ser.interpolate(limit_direction='both', limit_area='outside')
Out[48]:
0      5.0
1      5.0
2      5.0
3      NaN
4      NaN
5      NaN
6     13.0
7     13.0
8     13.0
dtype: float64

```

### 1.3.1.9 get\_dummies now supports dtype argument

The `get_dummies()` now accepts a `dtype` argument, which specifies a dtype for the new columns. The default remains `uint8`. (GH18330)

```

In [49]: df = pd.DataFrame({'a': [1, 2], 'b': [3, 4], 'c': [5, 6]})
In [50]: pd.get_dummies(df, columns=['c']).dtypes
Out[50]:
a      int64
b      int64
c_5    uint8
c_6    uint8
dtype: object

In [51]: pd.get_dummies(df, columns=['c'], dtype=bool).dtypes
Out[51]:
a      int64
b      int64
c_5    bool
c_6    bool
dtype: object

```

### 1.3.1.10 Timedelta mod method

`mod (%)` and `divmod` operations are now defined on `Timedelta` objects when operating with either `timedelta`-like or with numeric arguments. See the [documentation here](#). (GH19365)

```

In [52]: td = pd.Timedelta(hours=37)

```

(continues on next page)



(continued from previous page)

```
In [53]: td % pd.Timedelta(minutes=45)
Out [53]: Timedelta('0 days 00:15:00')
```

### 1.3.1.11 .rank() handles inf values when NaN are present

In previous versions, `.rank()` would assign `inf` elements `NaN` as their ranks. Now ranks are calculated properly. (GH6945)

```
In [54]: s = pd.Series([-np.inf, 0, 1, np.nan, np.inf])
In [55]: s
Out [55]:
0      -inf
1    0.000000
2    1.000000
3         NaN
4         inf
dtype: float64
```

Previous Behavior:

```
In [11]: s.rank()
Out [11]:
0      1.0
1      2.0
2      3.0
3      NaN
4      NaN
dtype: float64
```

Current Behavior:

```
In [56]: s.rank()
Out [56]:
0      1.0
1      2.0
2      3.0
3      NaN
4      4.0
dtype: float64
```

Furthermore, previously if you rank `inf` or `-inf` values together with `NaN` values, the calculation won't distinguish `NaN` from infinity when using 'top' or 'bottom' argument.

```
In [57]: s = pd.Series([np.nan, np.nan, -np.inf, -np.inf])
In [58]: s
Out [58]:
0      NaN
1      NaN
2     -inf
3     -inf
dtype: float64
```

Previous Behavior:

```
In [15]: s.rank(na_option='top')
Out[15]:
0    2.5
1    2.5
2    2.5
3    2.5
dtype: float64
```

Current Behavior:

```
In [59]: s.rank(na_option='top')
Out[59]:
0    1.5
1    1.5
2    3.5
3    3.5
dtype: float64
```

These bugs were squashed:

- Bug in `DataFrame.rank()` and `Series.rank()` when `method='dense'` and `pct=True` in which percentile ranks were not being used with the number of distinct observations ([GH15630](#))
- Bug in `Series.rank()` and `DataFrame.rank()` when `ascending=False` failed to return correct ranks for infinity if NaN were present ([GH19538](#))
- Bug in `DataFrameGroupBy.rank()` where ranks were incorrect when both infinity and NaN were present ([GH20561](#))

### 1.3.1.12 `Series.str.cat` has gained the `join` kwarg

Previously, `Series.str.cat()` did not – in contrast to most of pandas – align `Series` on their index before concatenation (see [GH18657](#)). The method has now gained a keyword `join` to control the manner of alignment, see examples below and [here](#).

In v.0.23 `join` will default to `None` (meaning no alignment), but this default will change to `'left'` in a future version of pandas.

```
In [60]: s = pd.Series(['a', 'b', 'c', 'd'])

In [61]: t = pd.Series(['b', 'd', 'e', 'c'], index=[1, 3, 4, 2])

In [62]: s.str.cat(t)
Out[62]:
0    ab
1    bd
2    ce
3    dc
dtype: object

In [63]: s.str.cat(t, join='left', na_rep='-')
Out[63]:
0    a-
1    bb
2    cc
3    dd
dtype: object
```

Furthermore, `Series.str.cat()` now works for `CategoricalIndex` as well (previously raised a `ValueError`; see [GH20842](#)).

### 1.3.1.13 DataFrame.astype performs column-wise conversion to Categorical

`DataFrame.astype()` can now perform column-wise conversion to `Categorical` by supplying the string `'category'` or a `CategoricalDtype`. Previously, attempting this would raise a `NotImplementedError`. See the *Object Creation* section of the documentation for more details and examples. ([GH12860](#), [GH18099](#))

Supplying the string `'category'` performs column-wise conversion, with only labels appearing in a given column set as categories:

```
In [64]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [65]: df = df.astype('category')

In [66]: df['A'].dtype
Out[66]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=False)

In [67]: df['B'].dtype
Out[67]: CategoricalDtype(categories=['b', 'c', 'd'], ordered=False)
```

Supplying a `CategoricalDtype` will make the categories in each column consistent with the supplied dtype:

```
In [68]: from pandas.api.types import CategoricalDtype

In [69]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})

In [70]: cdt = CategoricalDtype(categories=list('abcd'), ordered=True)

In [71]: df = df.astype(cdt)

In [72]: df['A'].dtype
Out[72]: CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)

In [73]: df['B'].dtype
Out[73]: CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)
```

### 1.3.1.14 Other Enhancements

- Unary `+` now permitted for `Series` and `DataFrame` as numeric operator ([GH16073](#))
- Better support for `to_excel()` output with the `xlsxwriter` engine. ([GH16149](#))
- `pandas.tseries.frequencies.to_offset()` now accepts leading `+` signs e.g. `‘+1h’`. ([GH18171](#))
- `MultiIndex.unique()` now supports the `level=` argument, to get unique values from a specific index level ([GH17896](#))
- `pandas.io.formats.style.Styler` now has method `hide_index()` to determine whether the index will be rendered in output ([GH14194](#))
- `pandas.io.formats.style.Styler` now has method `hide_columns()` to determine whether columns will be hidden in output ([GH14194](#))

- Improved wording of `ValueError` raised in `to_datetime()` when `unit=` is passed with a non-convertible value (GH14350)
- `Series.fillna()` now accepts a `Series` or a `dict` as a value for a categorical dtype (GH17033)
- `pandas.read_clipboard()` updated to use `qtpy`, falling back to `PyQt5` and then `PyQt4`, adding compatibility with `Python3` and multiple `python-qt` bindings (GH17722)
- Improved wording of `ValueError` raised in `read_csv()` when the `usecols` argument cannot match all columns. (GH17301)
- `DataFrame.corrwith()` now silently drops non-numeric columns when passed a `Series`. Before, an exception was raised (GH18570).
- `IntervalIndex` now supports time zone aware `Interval` objects (GH18537, GH18538)
- `Series()` / `DataFrame()` tab completion also returns identifiers in the first level of a `MultiIndex()`. (GH16326)
- `read_excel()` has gained the `nrows` parameter (GH16645)
- `DataFrame.append()` can now in more cases preserve the type of the calling dataframe's columns (e.g. if both are `CategoricalIndex`) (GH18359)
- `DataFrame.to_json()` and `Series.to_json()` now accept an `index` argument which allows the user to exclude the index from the JSON output (GH17394)
- `IntervalIndex.to_tuples()` has gained the `na_tuple` parameter to control whether NA is returned as a tuple of NA, or NA itself (GH18756)
- `Categorical.rename_categories`, `CategoricalIndex.rename_categories` and `Series.cat.rename_categories` can now take a callable as their argument (GH18862)
- `Interval` and `IntervalIndex` have gained a `length` attribute (GH18789)
- Resampler objects now have a functioning `pipe` method. Previously, calls to `pipe` were diverted to the `mean` method (GH17905).
- `is_scalar()` now returns `True` for `DateOffset` objects (GH18943).
- `DataFrame.pivot()` now accepts a list for the `values=` kwarg (GH17160).
- Added `pandas.api.extensions.register_dataframe_accessor()`, `pandas.api.extensions.register_series_accessor()`, and `pandas.api.extensions.register_index_accessor()`, accessor for libraries downstream of pandas to register custom accessors like `.cat` on pandas objects. See [Registering Custom Accessors](#) for more (GH14781).
- `IntervalIndex.astype` now supports conversions between subtypes when passed an `IntervalDtype` (GH19197)
- `IntervalIndex` and its associated constructor methods (`from_arrays`, `from_breaks`, `from_tuples`) have gained a `dtype` parameter (GH19262)
- Added `pandas.core.groupby.SeriesGroupBy.is_monotonic_increasing()` and `pandas.core.groupby.SeriesGroupBy.is_monotonic_decreasing()` (GH17015)
- For subclassed `DataFrames`, `DataFrame.apply()` will now preserve the `Series` subclass (if defined) when passing the data to the applied function (GH19822)
- `DataFrame.from_dict()` now accepts a `columns` argument that can be used to specify the column names when `orient='index'` is used (GH18529)
- Added option `display.html.use_mathjax` so `MathJax` can be disabled when rendering tables in Jupyter notebooks (GH19856, GH19824)

- `DataFrame.replace()` now supports the `method` parameter, which can be used to specify the replacement method when `to_replace` is a scalar, list or tuple and `value` is `None` (GH19632)
- `Timestamp.month_name()`, `DatetimeIndex.month_name()`, and `Series.dt.month_name()` are now available (GH12805)
- `Timestamp.day_name()` and `DatetimeIndex.day_name()` are now available to return day names with a specified locale (GH12806)
- `DataFrame.to_sql()` now performs a multivalue insert if the underlying connection supports itk rather than inserting row by row. SQLAlchemy dialects supporting multivalue inserts include: `mysql`, `postgresql`, `sqlite` and any dialect with `supports_multivalues_insert`. (GH14315, GH8953)
- `read_html()` now accepts a `displayed_only` keyword argument to controls whether or not hidden elements are parsed (True by default) (GH20027)
- `read_html()` now reads all `<tbody>` elements in a `<table>`, not just the first. (GH20690)
- `quantile()` and `quantile()` now accept the `interpolation` keyword, `linear` by default (GH20497)
- zip compression is supported via `compression=zip` in `DataFrame.to_pickle()`, `Series.to_pickle()`, `DataFrame.to_csv()`, `Series.to_csv()`, `DataFrame.to_json()`, `Series.to_json()`. (GH17778)
- `WeekOfMonth` constructor now supports `n=0` (GH20517).
- `DataFrame` and `Series` now support matrix multiplication (`@`) operator (GH10259) for `Python>=3.5`
- Updated `DataFrame.to_gbq()` and `pandas.read_gbq()` signature and documentation to reflect changes from the Pandas-GBQ library version 0.4.0. Adds intersphinx mapping to Pandas-GBQ library. (GH20564)
- Added new writer for exporting Stata dta files in version 117, `StataWriter117`. This format supports exporting strings up to 2,000,000 characters (GH16450)
- `to_hdf()` and `read_hdf()` now accept an `errors` keyword argument to control encoding error handling (GH20835)
- `cut()` has gained the `duplicates='raise'|'drop'` option to control whether to raise on duplicated edges (GH20947)
- `date_range()`, `timedelta_range()`, and `interval_range()` now return a linearly spaced index if `start`, `stop`, and `periods` are specified, but `freq` is not. (GH20808, GH20983, GH20976)

## 1.3.2 Backwards incompatible API changes

### 1.3.2.1 Dependencies have increased minimum versions

We have updated our minimum supported versions of dependencies (GH15184). If installed, we now require:

Package	Minimum Version	Required	Issue
python-dateutil	2.5.0	X	GH15184
openpyxl	2.4.0		GH15184
beautifulsoup4	4.2.1		GH20082
setuptools	24.2.0		GH20698

### 1.3.2.2 Instantiation from dicts preserves dict insertion order for python 3.6+

Until Python 3.6, dicts in Python had no formally defined ordering. For Python version 3.6 and later, dicts are ordered by insertion order, see [PEP 468](#). Pandas will use the dict's insertion order, when creating a `Series` or `DataFrame` from a dict and you're using Python version 3.6 or higher. ([GH19884](#))

Previous Behavior (and current behavior if on Python < 3.6):

```
pd.Series({'Income': 2000,
          'Expenses': -1500,
          'Taxes': -200,
          'Net result': 300})
Expenses    -1500
Income       2000
Net result    300
Taxes        -200
dtype: int64
```

Note the `Series` above is ordered alphabetically by the index values.

New Behavior (for Python >= 3.6):

```
In [74]: pd.Series({'Income': 2000,
.....:             'Expenses': -1500,
.....:             'Taxes': -200,
.....:             'Net result': 300})
Out[74]:
Income       2000
Expenses    -1500
Taxes        -200
Net result    300
dtype: int64
```

Notice that the `Series` is now ordered by insertion order. This new behavior is used for all relevant pandas types (`Series`, `DataFrame`, `SparseSeries` and `SparseDataFrame`).

If you wish to retain the old behavior while using Python >= 3.6, you can use `.sort_index()`:

```
In [75]: pd.Series({'Income': 2000,
.....:             'Expenses': -1500,
.....:             'Taxes': -200,
.....:             'Net result': 300}).sort_index()
Out[75]:
Expenses    -1500
Income       2000
Net result    300
Taxes        -200
dtype: int64
```

### 1.3.2.3 Deprecate Panel

`Panel` was deprecated in the 0.20.x release, showing as a `DeprecationWarning`. Using `Panel` will now show a `FutureWarning`. The recommended way to represent 3-D data are with a `MultiIndex` on a `DataFrame` via the `to_frame()` or with the `xarray` package. Pandas provides a `to_xarray()` method to automate this conversion. For more details see [Deprecate Panel](#) documentation. ([GH13563](#), [GH18324](#)).

```
In [76]: p = tm.makePanel()
```

```
In [77]: p
```

```
Out [77]:
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

### Convert to a MultiIndex DataFrame

```
In [78]: p.to_frame()
```

```
Out [78]:
```

		ItemA	ItemB	ItemC
major	minor			
2000-01-03	A	1.474071	-0.964980	-1.197071
	B	0.781836	1.846883	-0.858447
	C	2.353925	-1.717693	0.384316
	D	-0.744471	0.901805	0.476720
2000-01-04	A	-0.064034	-0.845696	-1.066969
	B	-1.071357	-1.328865	0.306996
	C	0.583787	0.888782	1.574159
	D	0.758527	1.171216	0.473424
2000-01-05	A	-1.282782	-1.340896	-0.303421
	B	0.441153	1.682706	-0.028665
	C	0.221471	0.228440	1.588931
	D	1.729689	0.520260	-0.242861

### Convert to an xarray DataArray

```
In [79]: p.to_xarray()
```

```
Out [79]:
```

```
<xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)>
array([[[ 1.474071,  0.781836,  2.353925, -0.744471],
        [-0.064034, -1.071357,  0.583787,  0.758527],
        [-1.282782,  0.441153,  0.221471,  1.729689]],

        [[-0.96498 ,  1.846883, -1.717693,  0.901805],
        [-0.845696, -1.328865,  0.888782,  1.171216],
        [-1.340896,  1.682706,  0.22844 ,  0.52026 ]],

        [[-1.197071, -0.858447,  0.384316,  0.47672 ],
        [-1.066969,  0.306996,  1.574159,  0.473424],
        [-0.303421, -0.028665,  1.588931, -0.242861]]])
Coordinates:
  * items          (items) object 'ItemA' 'ItemB' 'ItemC'
  * major_axis     (major_axis) datetime64[ns] 2000-01-03 2000-01-04 2000-01-05
  * minor_axis     (minor_axis) object 'A' 'B' 'C' 'D'
```

### 1.3.2.4 pandas.core.common removals

The following error & warning messages are removed from `pandas.core.common` ([GH13634](#), [GH19769](#)):

- `PerformanceWarning`
- `UnsupportedFunctionCall`

- `UnsortedIndexError`
- `AbstractMethodError`

These are available from import from `pandas.errors` (since 0.19.0).

### 1.3.2.5 Changes to make output of `DataFrame.apply` consistent

`DataFrame.apply()` was inconsistent when applying an arbitrary user-defined-function that returned a list-like with `axis=1`. Several bugs and inconsistencies are resolved. If the applied function returns a Series, then pandas will return a DataFrame; otherwise a Series will be returned, this includes the case where a list-like (e.g. tuple or list is returned) (GH16353, GH17437, GH17970, GH17348, GH17892, GH18573, GH17602, GH18775, GH18901, GH18919).

```
In [80]: df = pd.DataFrame(np.tile(np.arange(3), 6).reshape(6, -1) + 1, columns=['A',  
→ 'B', 'C'])
```

```
In [81]: df  
Out[81]:
```

	A	B	C
0	1	2	3
1	1	2	3
2	1	2	3
3	1	2	3
4	1	2	3
5	1	2	3

Previous Behavior: if the returned shape happened to match the length of original columns, this would return a DataFrame. If the return shape did not match, a Series with lists was returned.

```
In [3]: df.apply(lambda x: [1, 2, 3], axis=1)  
Out[3]:
```

	A	B	C
0	1	2	3
1	1	2	3
2	1	2	3
3	1	2	3
4	1	2	3
5	1	2	3

```
In [4]: df.apply(lambda x: [1, 2], axis=1)  
Out[4]:
```

0	[1, 2]
1	[1, 2]
2	[1, 2]
3	[1, 2]
4	[1, 2]
5	[1, 2]

dtype: object

New Behavior: When the applied function returns a list-like, this will now *always* return a Series.

```
In [82]: df.apply(lambda x: [1, 2, 3], axis=1)
```

```
Out[82]:  
0    [1, 2, 3]  
1    [1, 2, 3]  
2    [1, 2, 3]
```

(continues on next page)



(continued from previous page)

```
3      [1, 2, 3]
4      [1, 2, 3]
5      [1, 2, 3]
dtype: object
```

```
In [83]: df.apply(lambda x: [1, 2], axis=1)
```

```
////////////////////////////////////
```

```
↪
0      [1, 2]
1      [1, 2]
2      [1, 2]
3      [1, 2]
4      [1, 2]
5      [1, 2]
dtype: object
```

To have expanded columns, you can use `result_type='expand'`

```
In [84]: df.apply(lambda x: [1, 2, 3], axis=1, result_type='expand')
```

```
Out [84]:
   0  1  2
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3
```

To broadcast the result across the original columns (the old behaviour for list-likes of the correct length), you can use `result_type='broadcast'`. The shape must match the original columns.

```
In [85]: df.apply(lambda x: [1, 2, 3], axis=1, result_type='broadcast')
```

```
Out [85]:
   A  B  C
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3
```

Returning a `Series` allows one to control the exact return structure and column names:

```
In [86]: df.apply(lambda x: Series([1, 2, 3], index=['D', 'E', 'F']), axis=1)
```

```
Out [86]:
   D  E  F
0  1  2  3
1  1  2  3
2  1  2  3
3  1  2  3
4  1  2  3
5  1  2  3
```

### 1.3.2.6 Concatenation will no longer sort

In a future version of pandas `pandas.concat()` will no longer sort the non-concatenation axis when it is not already aligned. The current behavior is the same as the previous (sorting), but now a warning is issued when `sort` is not specified and the non-concatenation axis is not aligned (GH4588).

```
In [87]: df1 = pd.DataFrame({"a": [1, 2], "b": [1, 2]}, columns=['b', 'a'])
In [88]: df2 = pd.DataFrame({"a": [4, 5]})
In [89]: pd.concat([df1, df2])
Out[89]:
```

	a	b
0	1	1.0
1	2	2.0
0	4	NaN
1	5	NaN

To keep the previous behavior (sorting) and silence the warning, pass `sort=True`

```
In [90]: pd.concat([df1, df2], sort=True)
Out[90]:
```

	a	b
0	1	1.0
1	2	2.0
0	4	NaN
1	5	NaN

To accept the future behavior (no sorting), pass `sort=False`

Note that this change also applies to `DataFrame.append()`, which has also received a `sort` keyword for controlling this behavior.

### 1.3.2.7 Build Changes

- Building pandas for development now requires `cython >= 0.24` (GH18613)
- Building from source now explicitly requires `setuptools` in `setup.py` (GH18113)
- Updated conda recipe to be in compliance with conda-build 3.0+ (GH18002)

### 1.3.2.8 Index Division By Zero Fills Correctly

Division operations on `Index` and subclasses will now fill division of positive numbers by zero with `np.inf`, division of negative numbers by zero with `-np.inf` and `0 / 0` with `np.nan`. This matches existing `Series` behavior. (GH19322, GH19347)

Previous Behavior:

```
In [6]: index = pd.Int64Index([-1, 0, 1])
In [7]: index / 0
Out[7]: Int64Index([0, 0, 0], dtype='int64')

# Previous behavior yielded different results depending on the type of zero in the_
↳divisor
```

(continues on next page)

(continued from previous page)

```

In [8]: index / 0.0
Out[8]: Float64Index([-inf, nan, inf], dtype='float64')

In [9]: index = pd.UInt64Index([0, 1])

In [10]: index / np.array([0, 0], dtype=np.uint64)
Out[10]: UInt64Index([0, 0], dtype='uint64')

In [11]: pd.RangeIndex(1, 5) / 0
ZeroDivisionError: integer division or modulo by zero

```

**Current Behavior:**

```

In [91]: index = pd.Int64Index([-1, 0, 1])

# division by zero gives -infinity where negative, +infinity where positive, and NaN
↳ for 0 / 0
In [92]: index / 0
Out[92]: Float64Index([-inf, nan, inf], dtype='float64')

# The result of division by zero should not depend on whether the zero is int or float
In [93]: index / 0.0
Out[93]: Float64Index([-inf, nan, inf], dtype='float64')

In [94]: index = pd.UInt64Index([0, 1])

In [95]: index / np.array([0, 0], dtype=np.uint64)
Out[95]: Float64Index([nan, inf], dtype='float64')

In [96]: pd.RangeIndex(1, 5) / 0
Out[96]: Float64Index([inf, inf, inf, inf], dtype='float64')

```

**1.3.2.9 Extraction of matching patterns from strings**

By default, extracting matching patterns from strings with `str.extract()` used to return a Series if a single group was being extracted (a DataFrame if more than one group was extracted). As of Pandas 0.23.0 `str.extract()` always returns a DataFrame, unless `expand` is set to `False`. Finally, `None` was an accepted value for the `expand` parameter (which was equivalent to `False`), but now raises a `ValueError`. ([GH11386](#))

**Previous Behavior:**

```

In [1]: s = pd.Series(['number 10', '12 eggs'])

In [2]: extracted = s.str.extract('.*(\d\d).*')

In [3]: extracted
Out [3]:
0    10
1    12
dtype: object

In [4]: type(extracted)
Out [4]:
pandas.core.series.Series

```

New Behavior:

```
In [97]: s = pd.Series(['number 10', '12 eggs'])

In [98]: extracted = s.str.extract('.*(\d\d).*')

In [99]: extracted
Out[99]:
      0
0   10
1   12

In [100]: type(extracted)
Out[100]: pandas.core.frame.DataFrame
```

To restore previous behavior, simply set `expand` to `False`:

```
In [101]: s = pd.Series(['number 10', '12 eggs'])

In [102]: extracted = s.str.extract('.*(\d\d).*', expand=False)

In [103]: extracted
Out[103]:
0    10
1    12
dtype: object

In [104]: type(extracted)
Out[104]: pandas.core.series.Series
```

### 1.3.2.10 Default value for the `ordered` parameter of `CategoricalDtype`

The default value of the `ordered` parameter for `CategoricalDtype` has changed from `False` to `None` to allow updating of categories without impacting `ordered`. Behavior should remain consistent for downstream objects, such as `Categorical` (GH18790)

In previous versions, the default value for the `ordered` parameter was `False`. This could potentially lead to the `ordered` parameter unintentionally being changed from `True` to `False` when users attempt to update categories if `ordered` is not explicitly specified, as it would silently default to `False`. The new behavior for `ordered=None` is to retain the existing value of `ordered`.

New Behavior:

```
In [105]: from pandas.api.types import CategoricalDtype

In [106]: cat = pd.Categorical(list('abcaba'), ordered=True, categories=list('cba'))

In [107]: cat
Out[107]:
[a, b, c, a, b, a]
Categories (3, object): [c < b < a]

In [108]: cdt = CategoricalDtype(categories=list('cbad'))

In [109]: cat.astype(cdt)
Out[109]:
```

(continues on next page)

(continued from previous page)

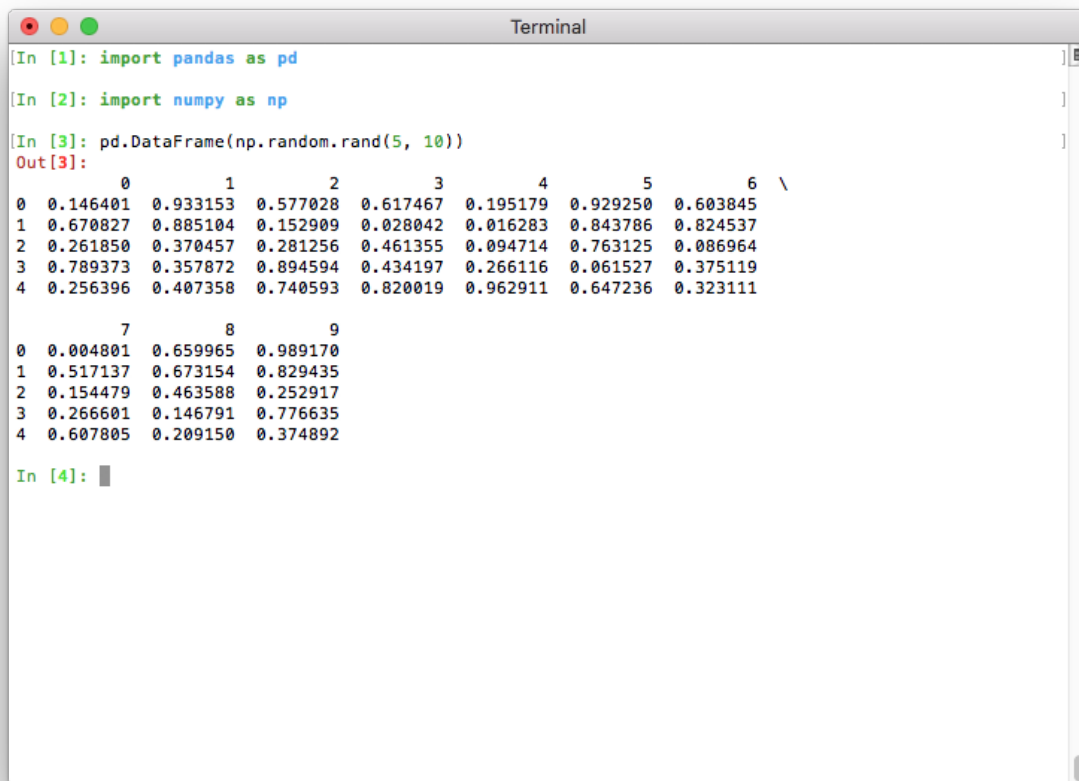
```
[a, b, c, a, b, a]
Categories (4, object): [c < b < a < d]
```

Notice in the example above that the converted Categorical has retained `ordered=True`. Had the default value for `ordered` remained as `False`, the converted Categorical would have become `unordered`, despite `ordered=False` never being explicitly specified. To change the value of `ordered`, explicitly pass it to the new dtype, e.g. `CategoricalDtype(categories=list('cbad'), ordered=False)`.

Note that the unintentional conversion of `ordered` discussed above did not arise in previous versions due to separate bugs that prevented `astype` from doing any type of category to category conversion ([GH10696](#), [GH18593](#)). These bugs have been fixed in this release, and motivated changing the default value of `ordered`.

### 1.3.2.11 Better pretty-printing of DataFrames in a terminal

Previously, the default value for the maximum number of columns was `pd.options.display.max_columns=20`. This meant that relatively wide data frames would not fit within the terminal width, and pandas would introduce line breaks to display these 20 columns. This resulted in an output that was relatively difficult to read:



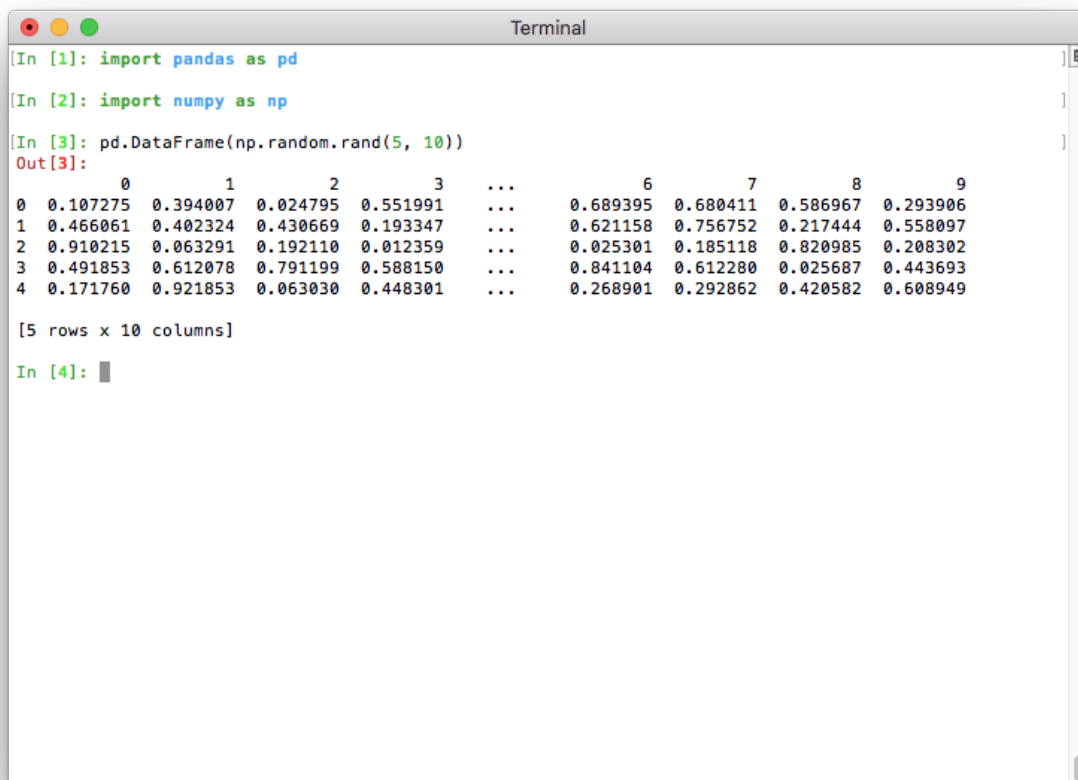
```
Terminal
[In [1]: import pandas as pd
[In [2]: import numpy as np
[In [3]: pd.DataFrame(np.random.rand(5, 10))
Out[3]:
```

	0	1	2	3	4	5	6	7	8	9
0	0.146401	0.933153	0.577028	0.617467	0.195179	0.929250	0.603845	0.004801	0.659965	0.989170
1	0.670827	0.885104	0.152909	0.028042	0.016283	0.843786	0.824537	0.517137	0.673154	0.829435
2	0.261850	0.370457	0.281256	0.461355	0.094714	0.763125	0.086964	0.154479	0.463588	0.252917
3	0.789373	0.357872	0.894594	0.434197	0.266116	0.061527	0.375119	0.266601	0.146791	0.776635
4	0.256396	0.407358	0.740593	0.820019	0.962911	0.647236	0.323111	0.607805	0.209150	0.374892

```

In [4]:
```

If Python runs in a terminal, the maximum number of columns is now determined automatically so that the printed data frame fits within the current terminal width (`pd.options.display.max_columns=0`) ([GH17023](#)). If Python runs as a Jupyter kernel (such as the Jupyter QtConsole or a Jupyter notebook, as well as in many IDEs), this value cannot be inferred automatically and is thus set to 20 as in previous versions. In a terminal, this results in a much nicer output:



```

Terminal
[In [1]: import pandas as pd
[In [2]: import numpy as np
[In [3]: pd.DataFrame(np.random.rand(5, 10))
Out[3]:
   0      1      2      3  ...      6      7      8      9
0  0.107275  0.394007  0.024795  0.551991  ...  0.689395  0.680411  0.586967  0.293906
1  0.466061  0.402324  0.430669  0.193347  ...  0.621158  0.756752  0.217444  0.558097
2  0.910215  0.063291  0.192110  0.012359  ...  0.025301  0.185118  0.820985  0.208302
3  0.491853  0.612078  0.791199  0.588150  ...  0.841104  0.612280  0.025687  0.443693
4  0.171760  0.921853  0.063030  0.448301  ...  0.268901  0.292862  0.420582  0.608949

[5 rows x 10 columns]
In [4]:

```

Note that if you don't like the new default, you can always set this option yourself. To revert to the old setting, you can run this line:

```
pd.options.display.max_columns = 20
```

### 1.3.2.12 Datetimelike API Changes

- The default `Timedelta` constructor now accepts an ISO 8601 Duration string as an argument (GH19040)
- Subtracting `NaT` from a `Series` with `dtype='datetime64[ns]'` returns a `Series` with `dtype='timedelta64[ns]'` instead of `dtype='datetime64[ns]'` (GH18808)
- Addition or subtraction of `NaT` from `TimedeltaIndex` will return `TimedeltaIndex` instead of `DatetimeIndex` (GH19124)
- `DatetimeIndex.shift()` and `TimedeltaIndex.shift()` will now raise `NullFrequencyError` (which subclasses `ValueError`, which was raised in older versions) when the index object frequency is `None` (GH19147)
- Addition and subtraction of `NaN` from a `Series` with `dtype='timedelta64[ns]'` will raise a `TypeError` instead of treating the `NaN` as `NaT` (GH19274)
- `NaT` division with `datetime.timedelta` will now return `NaN` instead of raising (GH17876)

- Operations between a *Series* with dtype `dtype='datetime64[ns]'` and a *PeriodIndex* will correctly raise `TypeError` (GH18850)
- Subtraction of *Series* with timezone-aware `dtype='datetime64[ns]'` with mis-matched timezones will raise `TypeError` instead of `ValueError` (GH18817)
- *Timestamp* will no longer silently ignore unused or invalid `tz` or `tzinfo` keyword arguments (GH17690)
- *Timestamp* will no longer silently ignore invalid `freq` arguments (GH5168)
- `CacheableOffset` and `WeekDay` are no longer available in the `pandas.tseries.offsets` module (GH17830)
- `pandas.tseries.frequencies.get_freq_group()` and `pandas.tseries.frequencies.DAYS` are removed from the public API (GH18034)
- *Series.truncate()* and *DataFrame.truncate()* will raise a `ValueError` if the index is not sorted instead of an unhelpful `KeyError` (GH17935)
- *Series.first* and *DataFrame.first* will now raise a `TypeError` rather than `NotImplementedError` when index is not a *DatetimeIndex* (GH20725).
- *Series.last* and *DataFrame.last* will now raise a `TypeError` rather than `NotImplementedError` when index is not a *DatetimeIndex* (GH20725).
- Restricted `DateOffset` keyword arguments. Previously, `DateOffset` subclasses allowed arbitrary keyword arguments which could lead to unexpected behavior. Now, only valid arguments will be accepted. (GH17176, GH18226).
- *pandas.merge()* provides a more informative error message when trying to merge on timezone-aware and timezone-naive columns (GH15800)
- For *DatetimeIndex* and *TimedeltaIndex* with `freq=None`, addition or subtraction of integer-dtyped array or `Index` will raise `NullFrequencyError` instead of `TypeError` (GH19895)
- *Timestamp* constructor now accepts a `nanosecond` keyword or positional argument (GH18898)
- *DatetimeIndex* will now raise an `AttributeError` when the `tz` attribute is set after instantiation (GH3746)
- *DatetimeIndex* with a `pytz` timezone will now return a consistent `pytz` timezone (GH18595)

### 1.3.2.13 Other API Changes

- *Series.astype()* and *Index.astype()* with an incompatible dtype will now raise a `TypeError` rather than a `ValueError` (GH18231)
- *Series* construction with an object dtyped tz-aware datetime and `dtype=object` specified, will now return an object dtyped *Series*, previously this would infer the datetime dtype (GH18231)
- A *Series* of `dtype=category` constructed from an empty dict will now have categories of `dtype=object` rather than `dtype=float64`, consistently with the case in which an empty list is passed (GH18515)
- All-`NaN` levels in a *MultiIndex* are now assigned `float` rather than `object` dtype, promoting consistency with *Index* (GH17929).
- Levels names of a *MultiIndex* (when not `None`) are now required to be unique: trying to create a *MultiIndex* with repeated names will raise a `ValueError` (GH18872)
- Both construction and renaming of *Index*/*MultiIndex* with non-hashable name/names will now raise `TypeError` (GH20527)

- `Index.map()` can now accept `Series` and dictionary input objects (GH12756, GH18482, GH18509).
- `DataFrame.unstack()` will now default to filling with `np.nan` for object columns. (GH12815)
- `IntervalIndex` constructor will raise if the `closed` parameter conflicts with how the input data is inferred to be closed (GH18421)
- Inserting missing values into indexes will work for all types of indexes and automatically insert the correct type of missing value (`NaN`, `NaT`, etc.) regardless of the type passed in (GH18295)
- When created with duplicate labels, `MultiIndex` now raises a `ValueError`. (GH17464)
- `Series.fillna()` now raises a `TypeError` instead of a `ValueError` when passed a list, tuple or `DataFrame` as a value (GH18293)
- `pandas.DataFrame.merge()` no longer casts a float column to object when merging on int and float columns (GH16572)
- `pandas.merge()` now raises a `ValueError` when trying to merge on incompatible data types (GH9780)
- The default NA value for `UInt64Index` has changed from 0 to `NaN`, which impacts methods that mask with NA, such as `UInt64Index.where()` (GH18398)
- Refactored `setup.py` to use `find_packages` instead of explicitly listing out all subpackages (GH18535)
- Rearranged the order of keyword arguments in `read_excel()` to align with `read_csv()` (GH16672)
- `wide_to_long()` previously kept numeric-like suffixes as object dtype. Now they are cast to numeric if possible (GH17627)
- In `read_excel()`, the `comment` argument is now exposed as a named parameter (GH18735)
- Rearranged the order of keyword arguments in `read_excel()` to align with `read_csv()` (GH16672)
- The options `html.border` and `mode.use_inf_as_null` were deprecated in prior versions, these will now show `FutureWarning` rather than a `DeprecationWarning` (GH19003)
- `IntervalIndex` and `IntervalDtype` no longer support categorical, object, and string subtypes (GH19016)
- `IntervalDtype` now returns `True` when compared against 'interval' regardless of subtype, and `IntervalDtype.name` now returns 'interval' regardless of subtype (GH18980)
- `KeyError` now raises instead of `ValueError` in `drop()`, `drop()`, `drop()`, `drop()` when dropping a non-existent element in an axis with duplicates (GH19186)
- `Series.to_csv()` now accepts a compression argument that works in the same way as the compression argument in `DataFrame.to_csv()` (GH18958)
- Set operations (union, difference...) on `IntervalIndex` with incompatible index types will now raise a `TypeError` rather than a `ValueError` (GH19329)
- `DateOffset` objects render more simply, e.g. `<DateOffset: days=1>` instead of `<DateOffset: kwds={'days': 1}>` (GH19403)
- `Categorical.fillna` now validates its value and method keyword arguments. It now raises when both or none are specified, matching the behavior of `Series.fillna()` (GH19682)
- `pd.to_datetime('today')` now returns a `datetime`, consistent with `pd.Timestamp('today')`; previously `pd.to_datetime('today')` returned a `.normalized()` `datetime` (GH19935)
- `Series.str.replace()` now takes an optional `regex` keyword which, when set to `False`, uses literal string replacement rather than regex replacement (GH16808)
- `DatetimeIndex.strftime()` and `PeriodIndex.strftime()` now return an `Index` instead of a numpy array to be consistent with similar accessors (GH20127)



- Constructing a Series from a list of length 1 no longer broadcasts this list when a longer index is specified ([GH19714](#), [GH20391](#)).
- `DataFrame.to_dict()` with `orient='index'` no longer casts int columns to float for a DataFrame with only int and float columns ([GH18580](#)).
- A user-defined-function that is passed to `Series.rolling().aggregate()`, `DataFrame.rolling().aggregate()`, or its expanding cousins, will now *always* be passed a Series, rather than a `np.array`; `.apply()` only has the `raw` keyword, see [here](#). This is consistent with the signatures of `.aggregate()` across pandas ([GH20584](#)).
- Rolling and Expanding types raise `NotImplementedError` upon iteration ([GH11704](#)).

### 1.3.3 Deprecations

- `Series.from_array` and `SparseSeries.from_array` are deprecated. Use the normal constructor `Series(...)` and `SparseSeries(...)` instead ([GH18213](#)).
- `DataFrame.as_matrix` is deprecated. Use `DataFrame.values` instead ([GH18458](#)).
- `Series.asobject`, `DatetimeIndex.asobject`, `PeriodIndex.asobject` and `TimeDeltaIndex.asobject` have been deprecated. Use `.astype(object)` instead ([GH18572](#)).
- Grouping by a tuple of keys now emits a `FutureWarning` and is deprecated. In the future, a tuple passed to 'by' will always refer to a single key that is the actual tuple, instead of treating the tuple as multiple keys. To retain the previous behavior, use a list instead of a tuple ([GH18314](#)).
- `Series.valid` is deprecated. Use `Series.dropna()` instead ([GH18800](#)).
- `read_excel()` has deprecated the `skip_footer` parameter. Use `skipfooter` instead ([GH18836](#)).
- `ExcelFile.parse()` has deprecated `sheetname` in favor of `sheet_name` for consistency with `read_excel()` ([GH20920](#)).
- The `is_copy` attribute is deprecated and will be removed in a future version ([GH18801](#)).
- `IntervalIndex.from_intervals` is deprecated in favor of the `IntervalIndex` constructor ([GH19263](#)).
- `DataFrame.from_items` is deprecated. Use `DataFrame.from_dict()` instead, or `DataFrame.from_dict(OrderedDict())` if you wish to preserve the key order ([GH17320](#), [GH17312](#)).
- Indexing a `MultiIndex` or a `FloatIndex` with a list containing some missing keys will now show a `FutureWarning`, which is consistent with other types of indexes ([GH17758](#)).
- The `broadcast` parameter of `.apply()` is deprecated in favor of `result_type='broadcast'` ([GH18577](#)).
- The `reduce` parameter of `.apply()` is deprecated in favor of `result_type='reduce'` ([GH18577](#)).
- The `order` parameter of `factorize()` is deprecated and will be removed in a future release ([GH19727](#)).
- `Timestamp.weekday_name`, `DatetimeIndex.weekday_name`, and `Series.dt.weekday_name` are deprecated in favor of `Timestamp.day_name()`, `DatetimeIndex.day_name()`, and `Series.dt.day_name()` ([GH12806](#)).
- `pandas.tseries.plotting.tsplot` is deprecated. Use `Series.plot()` instead ([GH18627](#)).
- `Index.summary()` is deprecated and will be removed in a future version ([GH18217](#)).
- `NDFrame.get_ftype_counts()` is deprecated and will be removed in a future version ([GH18243](#)).

- The `convert_datetime64` parameter in `DataFrame.to_records()` has been deprecated and will be removed in a future version. The NumPy bug motivating this parameter has been resolved. The default value for this parameter has also changed from `True` to `None` (GH18160).
- `Series.rolling().apply()`, `DataFrame.rolling().apply()`, `Series.expanding().apply()`, and `DataFrame.expanding().apply()` have deprecated passing an `np.array` by default. One will need to pass the new `raw` parameter to be explicit about what is passed (GH20584)
- The `data`, `base`, `strides`, `flags` and `itemsize` properties of the `Series` and `Index` classes have been deprecated and will be removed in a future version (GH20419).
- `DatetimeIndex.offset` is deprecated. Use `DatetimeIndex.freq` instead (GH20716)
- Floor division between an integer ndarray and a `Timedelta` is deprecated. Divide by `Timedelta.value` instead (GH19761)
- Setting `PeriodIndex.freq` (which was not guaranteed to work correctly) is deprecated. Use `PeriodIndex.asfreq()` instead (GH20678)
- `Index.get_duplicates()` is deprecated and will be removed in a future version (GH20239)
- The previous default behavior of negative indices in `Categorical.take` is deprecated. In a future version it will change from meaning missing values to meaning positional indices from the right. The future behavior is consistent with `Series.take()` (GH20664).
- Passing multiple axes to the `axis` parameter in `DataFrame.dropna()` has been deprecated and will be removed in a future version (GH20987)

### 1.3.4 Removal of prior version deprecations/changes

- Warnings against the obsolete usage `Categorical(codes, categories)`, which were emitted for instance when the first two arguments to `Categorical()` had different dtypes, and recommended the use of `Categorical.from_codes`, have now been removed (GH8074)
- The `levels` and `labels` attributes of a `MultiIndex` can no longer be set directly (GH4039).
- `pd.tseries.util.pivot_annual` has been removed (deprecated since v0.19). Use `pivot_table` instead (GH18370)
- `pd.tseries.util.isleapyear` has been removed (deprecated since v0.19). Use `.is_leap_year` property in `Datetime`-likes instead (GH18370)
- `pd.ordered_merge` has been removed (deprecated since v0.19). Use `pd.merge_ordered` instead (GH18459)
- The `SparseList` class has been removed (GH14007)
- The `pandas.io.wb` and `pandas.io.data` stub modules have been removed (GH13735)
- `Categorical.from_array` has been removed (GH13854)
- The `freq` and `how` parameters have been removed from the `rolling/expanding/ewm` methods of `DataFrame` and `Series` (deprecated since v0.18). Instead, `resample` before calling the methods. (GH18601 & GH18668)
- `DatetimeIndex.to_datetime`, `Timestamp.to_datetime`, `PeriodIndex.to_datetime`, and `Index.to_datetime` have been removed (GH8254, GH14096, GH14113)
- `read_csv()` has dropped the `skip_footer` parameter (GH13386)
- `read_csv()` has dropped the `as_reccarray` parameter (GH13373)
- `read_csv()` has dropped the `buffer_lines` parameter (GH13360)

- `read_csv()` has dropped the `compact_ints` and `use_unsigned` parameters (GH13323)
- The `Timestamp` class has dropped the `offset` attribute in favor of `freq` (GH13593)
- The `Series`, `Categorical`, and `Index` classes have dropped the `reshape` method (GH13012)
- `pandas.tseries.frequencies.get_standard_freq` has been removed in favor of `pandas.tseries.frequencies.to_offset(freq).rule_code` (GH13874)
- The `freqstr` keyword has been removed from `pandas.tseries.frequencies.to_offset` in favor of `freq` (GH13874)
- The `Panel4D` and `PanelND` classes have been removed (GH13776)
- The `Panel` class has dropped the `to_long` and `toLong` methods (GH19077)
- The options `display.line_with` and `display.height` are removed in favor of `display.width` and `display.max_rows` respectively (GH4391, GH19107)
- The `labels` attribute of the `Categorical` class has been removed in favor of `Categorical.codes` (GH7768)
- The `flavor` parameter have been removed from func:`to_sql` method (GH13611)
- The modules `pandas.tools.hashing` and `pandas.util.hashing` have been removed (GH16223)
- The top-level functions `pd.rolling_*`, `pd.expanding_*` and `pd.ewm*` have been removed (Deprecated since v0.18). Instead, use the `DataFrame/Series` methods `rolling`, `expanding` and `ewm` (GH18723)
- Imports from `pandas.core.common` for functions such as `is_datetime64_dtype` are now removed. These are located in `pandas.api.types`. (GH13634, GH19769)
- The `infer_dst` keyword in `Series.tz_localize()`, `DatetimeIndex.tz_localize()` and `DatetimeIndex` have been removed. `infer_dst=True` is equivalent to `ambiguous='infer'`, and `infer_dst=False` to `ambiguous='raise'` (GH7963).
- When `.resample()` was changed from an eager to a lazy operation, like `.groupby()` in v0.18.0, we put in place compatibility (with a `FutureWarning`), so operations would continue to work. This is now fully removed, so a `Resampler` will no longer forward compat operations (GH20554)
- Remove long deprecated `axis=None` parameter from `.replace()` (GH20271)

### 1.3.5 Performance Improvements

- Indexers on `Series` or `DataFrame` no longer create a reference cycle (GH17956)
- Added a keyword argument, `cache`, to `to_datetime()` that improved the performance of converting duplicate datetime arguments (GH11665)
- `DateOffset` arithmetic performance is improved (GH18218)
- Converting a `Series` of `Timedelta` objects to days, seconds, etc... sped up through vectorization of underlying methods (GH18092)
- Improved performance of `.map()` with a `Series/dict` input (GH15081)
- The overridden `Timedelta` properties of days, seconds and microseconds have been removed, leveraging their built-in Python versions instead (GH18242)
- `Series` construction will reduce the number of copies made of the input data in certain cases (GH17449)
- Improved performance of `Series.dt.date()` and `DatetimeIndex.date()` (GH18058)
- Improved performance of `Series.dt.time()` and `DatetimeIndex.time()` (GH18461)

- Improved performance of `IntervalIndex.symmetric_difference()` (GH18475)
- Improved performance of `DatetimeIndex` and `Series` arithmetic operations with `Business-Month` and `Business-Quarter` frequencies (GH18489)
- `Series()` / `DataFrame()` tab completion limits to 100 values, for better performance. (GH18587)
- Improved performance of `DataFrame.median()` with `axis=1` when `bottleneck` is not installed (GH16468)
- Improved performance of `MultiIndex.get_loc()` for large indexes, at the cost of a reduction in performance for small ones (GH18519)
- Improved performance of `MultiIndex.remove_unused_levels()` when there are no unused levels, at the cost of a reduction in performance when there are (GH19289)
- Improved performance of `Index.get_loc()` for non-unique indexes (GH19478)
- Improved performance of pairwise `.rolling()` and `.expanding()` with `.cov()` and `.corr()` operations (GH17917)
- Improved performance of `pandas.core.groupby.GroupBy.rank()` (GH15779)
- Improved performance of variable `.rolling()` on `.min()` and `.max()` (GH19521)
- Improved performance of `pandas.core.groupby.GroupBy.ffill()` and `pandas.core.groupby.GroupBy.bfill()` (GH11296)
- Improved performance of `pandas.core.groupby.GroupBy.any()` and `pandas.core.groupby.GroupBy.all()` (GH15435)
- Improved performance of `pandas.core.groupby.GroupBy.pct_change()` (GH19165)
- Improved performance of `Series.isin()` in the case of categorical dtypes (GH20003)
- Improved performance of `getattr(Series, attr)` when the `Series` has certain index types. This manifested in slow printing of large `Series` with a `DatetimeIndex` (GH19764)
- Fixed a performance regression for `GroupBy.nth()` and `GroupBy.last()` with some object columns (GH19283)
- Improved performance of `pandas.core.arrays.Categorical.from_codes()` (GH18501)

### 1.3.6 Documentation Changes

Thanks to all of the contributors who participated in the Pandas Documentation Sprint, which took place on March 10th. We had about 500 participants from over 30 locations across the world. You should notice that many of the *API docstrings* have greatly improved.

There were too many simultaneous contributions to include a release note for each improvement, but this [GitHub search](#) should give you an idea of how many docstrings were improved.

Special thanks to [Marc Garcia](#) for organizing the sprint. For more information, read the [NumFOCUS blogpost](#) recapping the sprint.

- Changed spelling of “numpy” to “NumPy”, and “python” to “Python”. (GH19017)
- Consistency when introducing code samples, using either colon or period. Rewrote some sentences for greater clarity, added more dynamic references to functions, methods and classes. (GH18941, GH18948, GH18973, GH19017)
- Added a reference to `DataFrame.assign()` in the concatenate section of the merging documentation (GH18665)

## 1.3.7 Bug Fixes

### 1.3.7.1 Categorical

**Warning:** A class of bugs were introduced in pandas 0.21 with `CategoricalDtype` that affects the correctness of operations like `merge`, `concat`, and indexing when comparing multiple unordered `Categorical` arrays that have the same categories, but in a different order. We highly recommend upgrading or manually aligning your categories before doing these operations.

- Bug in `Categorical.equals` returning the wrong result when comparing two unordered `Categorical` arrays with the same categories, but in a different order ([GH16603](#))
- Bug in `pandas.api.types.union_categoricals()` returning the wrong result when for unordered categoricals with the categories in a different order. This affected `pandas.concat()` with `Categorical` data ([GH19096](#)).
- Bug in `pandas.merge()` returning the wrong result when joining on an unordered `Categorical` that had the same categories but in a different order ([GH19551](#))
- Bug in `CategoricalIndex.get_indexer()` returning the wrong result when target was an unordered `Categorical` that had the same categories as `self` but in a different order ([GH19551](#))
- Bug in `Index.astype()` with a categorical dtype where the resultant index is not converted to a `CategoricalIndex` for all types of index ([GH18630](#))
- Bug in `Series.astype()` and `Categorical.astype()` where an existing categorical data does not get updated ([GH10696](#), [GH18593](#))
- Bug in `Series.str.split()` with `expand=True` incorrectly raising an `IndexError` on empty strings ([GH20002](#)).
- Bug in `Index` constructor with `dtype=CategoricalDtype(...)` where categories and ordered are not maintained ([GH19032](#))
- Bug in `Series` constructor with scalar and `dtype=CategoricalDtype(...)` where categories and ordered are not maintained ([GH19565](#))
- Bug in `Categorical.__iter__` not converting to Python types ([GH19909](#))
- Bug in `pandas.factorize()` returning the unique codes for the uniques. This now returns a `Categorical` with the same dtype as the input ([GH19721](#))
- Bug in `pandas.factorize()` including an item for missing values in the uniques return value ([GH19721](#))
- Bug in `Series.take()` with categorical data interpreting `-1` in `indices` as missing value markers, rather than the last element of the `Series` ([GH20664](#))

### 1.3.7.2 Datetimelike

- Bug in `Series.__sub__()` subtracting a non-nanosecond `np.datetime64` object from a `Series` gave incorrect results ([GH7996](#))
- Bug in `DatetimeIndex`, `TimedeltaIndex` addition and subtraction of zero-dimensional integer arrays gave incorrect results ([GH19012](#))

- Bug in `DatetimeIndex` and `TimedeltaIndex` where adding or subtracting an array-like of `DateOffset` objects either raised `(np.array, pd.Index)` or broadcast incorrectly `(pd.Series)` (GH18849)
- Bug in `Series.__add__()` adding `Series` with dtype `timedelta64[ns]` to a timezone-aware `DatetimeIndex` incorrectly dropped timezone information (GH13905)
- Adding a `Period` object to a `datetime` or `Timestamp` object will now correctly raise a `TypeError` (GH17983)
- Bug in `Timestamp` where comparison with an array of `Timestamp` objects would result in a `RecursionError` (GH15183)
- Bug in `Series` floor-division where operating on a scalar `timedelta` raises an exception (GH18846)
- Bug in `DatetimeIndex` where the repr was not showing high-precision time values at the end of a day (e.g., `23:59:59.999999999`) (GH19030)
- Bug in `.astype()` to non-ns `timedelta` units would hold the incorrect dtype (GH19176, GH19223, GH12425)
- Bug in subtracting `Series` from `NaT` incorrectly returning `NaT` (GH19158)
- Bug in `Series.truncate()` which raises `TypeError` with a monotonic `PeriodIndex` (GH17717)
- Bug in `pct_change()` using periods and freq returned different length outputs (GH7292)
- Bug in comparison of `DatetimeIndex` against `None` or `datetime.date` objects raising `TypeError` for `==` and `!=` comparisons instead of all-False and all-True, respectively (GH19301)
- Bug in `Timestamp` and `to_datetime()` where a string representing a barely out-of-bounds timestamp would be incorrectly rounded down instead of raising `OutOfBoundsDatetime` (GH19382)
- Bug in `Timestamp.floor()` `DatetimeIndex.floor()` where time stamps far in the future and past were not rounded correctly (GH19206)
- Bug in `to_datetime()` where passing an out-of-bounds datetime with `errors='coerce'` and `utc=True` would raise `OutOfBoundsDatetime` instead of parsing to `NaT` (GH19612)
- Bug in `DatetimeIndex` and `TimedeltaIndex` addition and subtraction where name of the returned object was not always set consistently. (GH19744)
- Bug in `DatetimeIndex` and `TimedeltaIndex` addition and subtraction where operations with numpy arrays raised `TypeError` (GH19847)
- Bug in `DatetimeIndex` and `TimedeltaIndex` where setting the `freq` attribute was not fully supported (GH20678)

### 1.3.7.3 Timedelta

- Bug in `Timedelta.__mul__()` where multiplying by `NaT` returned `NaT` instead of raising a `TypeError` (GH19819)
- Bug in `Series` with dtype `'timedelta64[ns]'` where addition or subtraction of `TimedeltaIndex` had results cast to dtype `'int64'` (GH17250)
- Bug in `Series` with dtype `'timedelta64[ns]'` where addition or subtraction of `TimedeltaIndex` could return a `Series` with an incorrect name (GH19043)
- Bug in `Timedelta.__floordiv__()` and `Timedelta.__rfloordiv__()` dividing by many incompatible numpy objects was incorrectly allowed (GH18846)
- Bug where dividing a scalar `timedelta`-like object with `TimedeltaIndex` performed the reciprocal operation (GH19125)



- Bug in `TimedeltaIndex` where division by a `Series` would return a `TimedeltaIndex` instead of a `Series` (GH19042)
- Bug in `Timedelta.__add__()`, `Timedelta.__sub__()` where adding or subtracting a `np.timedelta64` object would return another `np.timedelta64` instead of a `Timedelta` (GH19738)
- Bug in `Timedelta.__floordiv__()`, `Timedelta.__rfloordiv__()` where operating with a `Tick` object would raise a `TypeError` instead of returning a numeric value (GH19738)
- Bug in `Period.asfreq()` where periods near `datetime(1, 1, 1)` could be converted incorrectly (GH19643, GH19834)
- Bug in `Timedelta.total_seconds()` causing precision errors, for example `Timedelta('30S').total_seconds()==30.000000000000004` (GH19458)
- Bug in `Timedelta.__rmod__()` where operating with a `numpy.timedelta64` returned a `timedelta64` object instead of a `Timedelta` (GH19820)
- Multiplication of `TimedeltaIndex` by `TimedeltaIndex` will now raise `TypeError` instead of raising `ValueError` in cases of length mis-match (GH19333)
- Bug in indexing a `TimedeltaIndex` with a `np.timedelta64` object which was raising a `TypeError` (GH20393)

#### 1.3.7.4 Timezones

- Bug in creating a `Series` from an array that contains both tz-naive and tz-aware values will result in a `Series` whose `dtype` is tz-aware instead of object (GH16406)
- Bug in comparison of timezone-aware `DatetimeIndex` against `NaT` incorrectly raising `TypeError` (GH19276)
- Bug in `DatetimeIndex.astype()` when converting between timezone aware dtypes, and converting from timezone aware to naive (GH18951)
- Bug in comparing `DatetimeIndex`, which failed to raise `TypeError` when attempting to compare timezone-aware and timezone-naive datetimelike objects (GH18162)
- Bug in localization of a naive, datetime string in a `Series` constructor with a `datetime64[ns, tz]` dtype (GH174151)
- `Timestamp.replace()` will now handle Daylight Savings transitions gracefully (GH18319)
- Bug in tz-aware `DatetimeIndex` where addition/subtraction with a `TimedeltaIndex` or array with `dtype='timedelta64[ns]'` was incorrect (GH17558)
- Bug in `DatetimeIndex.insert()` where inserting `NaT` into a timezone-aware index incorrectly raised (GH16357)
- Bug in `DataFrame` constructor, where tz-aware `DatetimeIndex` and a given column name will result in an empty `DataFrame` (GH19157)
- Bug in `Timestamp.tz_localize()` where localizing a timestamp near the minimum or maximum valid values could overflow and return a timestamp with an incorrect nanosecond value (GH12677)
- Bug when iterating over `DatetimeIndex` that was localized with fixed timezone offset that rounded nanosecond precision to microseconds (GH19603)
- Bug in `DataFrame.diff()` that raised an `IndexError` with tz-aware values (GH18578)
- Bug in `melt()` that converted tz-aware dtypes to tz-naive (GH15785)

- Bug in `DataFrame.count()` that raised an `ValueError`, if `DataFrame.dropna()` was called for a single column with timezone-aware values. (GH13407)

### 1.3.7.5 Offsets

- Bug in `WeekOfMonth` and `Week` where addition and subtraction did not roll correctly (GH18510, GH18672, GH18864)
- Bug in `WeekOfMonth` and `LastWeekOfMonth` where default keyword arguments for constructor raised `ValueError` (GH19142)
- Bug in `FY5253Quarter`, `LastWeekOfMonth` where rollback and rollforward behavior was inconsistent with addition and subtraction behavior (GH18854)
- Bug in `FY5253` where `datetime` addition and subtraction incremented incorrectly for dates on the year-end but not normalized to midnight (GH18854)
- Bug in `FY5253` where date offsets could incorrectly raise an `AssertionError` in arithmetic operations (GH14774)

### 1.3.7.6 Numeric

- Bug in `Series` constructor with an int or float list where specifying `dtype=str`, `dtype='str'` or `dtype='U'` failed to convert the data elements to strings (GH16605)
- Bug in `Index` multiplication and division methods where operating with a `Series` would return an `Index` object instead of a `Series` object (GH19042)
- Bug in the `DataFrame` constructor in which data containing very large positive or very large negative numbers was causing `OverflowError` (GH18584)
- Bug in `Index` constructor with `dtype='uint64'` where int-like floats were not coerced to `UInt64Index` (GH18400)
- Bug in `DataFrame` flex arithmetic (e.g. `df.add(other, fill_value=foo)`) with a `fill_value` other than `None` failed to raise `NotImplementedError` in corner cases where either the frame or other has length zero (GH19522)
- Multiplication and division of numeric-dtyped `Index` objects with timedelta-like scalars returns `TimedeltaIndex` instead of raising `TypeError` (GH19333)
- Bug where `NaN` was returned instead of `0` by `Series.pct_change()` and `DataFrame.pct_change()` when `fill_method` is not `None` (GH19873)

### 1.3.7.7 Strings

- Bug in `Series.str.get()` with a dictionary in the values and the index not in the keys, raising `KeyError` (GH20671)

### 1.3.7.8 Indexing

- Bug in `Index` construction from list of mixed type tuples (GH18505)
- Bug in `Index.drop()` when passing a list of both tuples and non-tuples (GH18304)
- Bug in `DataFrame.drop()`, `Panel.drop()`, `Series.drop()`, `Index.drop()` where no `KeyError` is raised when dropping a non-existent element from an axis that contains duplicates (GH19186)



- Bug in indexing a datetimelike Index that raised `ValueError` instead of `IndexError` (GH18386).
- `Index.to_series()` now accepts index and name kwargs (GH18699)
- `DatetimeIndex.to_series()` now accepts index and name kwargs (GH18699)
- Bug in indexing non-scalar value from Series having non-unique Index will return value flattened (GH17610)
- Bug in indexing with iterator containing only missing keys, which raised no error (GH20748)
- Fixed inconsistency in `.ix` between list and scalar keys when the index has integer dtype and does not include the desired keys (GH20753)
- Bug in `__setitem__` when indexing a `DataFrame` with a 2-d boolean ndarray (GH18582)
- Bug in `str.extractall` when there were no matches empty `Index` was returned instead of appropriate `MultiIndex` (GH19034)
- Bug in `IntervalIndex` where empty and purely NA data was constructed inconsistently depending on the construction method (GH18421)
- Bug in `IntervalIndex.symmetric_difference()` where the symmetric difference with a non-`IntervalIndex` did not raise (GH18475)
- Bug in `IntervalIndex` where set operations that returned an empty `IntervalIndex` had the wrong dtype (GH19101)
- Bug in `DataFrame.drop_duplicates()` where no `KeyError` is raised when passing in columns that don't exist on the `DataFrame` (GH19726)
- Bug in Index subclasses constructors that ignore unexpected keyword arguments (GH19348)
- Bug in `Index.difference()` when taking difference of an Index with itself (GH20040)
- Bug in `DataFrame.first_valid_index()` and `DataFrame.last_valid_index()` in presence of entire rows of NaNs in the middle of values (GH20499).
- Bug in `IntervalIndex` where some indexing operations were not supported for overlapping or non-monotonic uint64 data (GH20636)
- Bug in `Series.is_unique` where extraneous output in stderr is shown if Series contains objects with `__ne__` defined (GH20661)
- Bug in `.loc` assignment with a single-element list-like incorrectly assigns as a list (GH19474)
- Bug in partial string indexing on a Series/DataFrame with a monotonic decreasing `DatetimeIndex` (GH19362)
- Bug in performing in-place operations on a `DataFrame` with a duplicate Index (GH17105)
- Bug in `IntervalIndex.get_loc()` and `IntervalIndex.get_indexer()` when used with an `IntervalIndex` containing a single interval (GH17284, GH20921)
- Bug in `.loc` with a uint64 indexer (GH20722)

#### 1.3.7.9 MultiIndex

- Bug in `MultiIndex.__contains__()` where non-tuple keys would return `True` even if they had been dropped (GH19027)
- Bug in `MultiIndex.set_labels()` which would cause casting (and potentially clipping) of the new labels if the `level` argument is not 0 or a list like `[0, 1, ...]` (GH19057)

- Bug in `MultiIndex.get_level_values()` which would return an invalid index on level of ints with missing values (GH17924)
- Bug in `MultiIndex.unique()` when called on empty `MultiIndex` (GH20568)
- Bug in `MultiIndex.unique()` which would not preserve level names (GH20570)
- Bug in `MultiIndex.remove_unused_levels()` which would fill nan values (GH18417)
- Bug in `MultiIndex.from_tuples()` which would fail to take zipped tuples in python3 (GH18434)
- Bug in `MultiIndex.get_loc()` which would fail to automatically cast values between float and int (GH18818, GH15994)
- Bug in `MultiIndex.get_loc()` which would cast boolean to integer labels (GH19086)
- Bug in `MultiIndex.get_loc()` which would fail to locate keys containing NaN (GH18485)
- Bug in `MultiIndex.get_loc()` in large `MultiIndex`, would fail when levels had different dtypes (GH18520)
- Bug in indexing where nested indexers having only numpy arrays are handled incorrectly (GH19686)

### 1.3.7.10 I/O

- `read_html()` now rewinds seekable IO objects after parse failure, before attempting to parse with a new parser. If a parser errors and the object is non-seekable, an informative error is raised suggesting the use of a different parser (GH17975)
- `DataFrame.to_html()` now has an option to add an id to the leading `<table>` tag (GH8496)
- Bug in `read_msgpack()` with a non-existent file is passed in Python 2 (GH15296)
- Bug in `read_csv()` where a `MultiIndex` with duplicate columns was not being mangled appropriately (GH18062)
- Bug in `read_csv()` where missing values were not being handled properly when `keep_default_na=False` with dictionary `na_values` (GH19227)
- Bug in `read_csv()` causing heap corruption on 32-bit, big-endian architectures (GH20785)
- Bug in `read_sas()` where a file with 0 variables gave an `AttributeError` incorrectly. Now it gives an `EmptyDataError` (GH18184)
- Bug in `DataFrame.to_latex()` where pairs of braces meant to serve as invisible placeholders were escaped (GH18667)
- Bug in `DataFrame.to_latex()` where a NaN in a `MultiIndex` would cause an `IndexError` or incorrect output (GH14249)
- Bug in `DataFrame.to_latex()` where a non-string index-level name would result in an `AttributeError` (GH19981)
- Bug in `DataFrame.to_latex()` where the combination of an index name and the `index_names=False` option would result in incorrect output (GH18326)
- Bug in `DataFrame.to_latex()` where a `MultiIndex` with an empty string as its name would result in incorrect output (GH18669)
- Bug in `DataFrame.to_latex()` where missing space characters caused wrong escaping and produced non-valid latex in some cases (GH20859)
- Bug in `read_json()` where large numeric values were causing an `OverflowError` (GH18842)
- Bug in `DataFrame.to_parquet()` where an exception was raised if the write destination is S3 (GH19134)

- `Interval` now supported in `DataFrame.to_excel()` for all Excel file types (GH19242)
- `Timedelta` now supported in `DataFrame.to_excel()` for all Excel file types (GH19242, GH9155, GH19900)
- Bug in `pandas.io.stata.StataReader.value_labels()` raising an `AttributeError` when called on very old files. Now returns an empty dict (GH19417)
- Bug in `read_pickle()` when unpickling objects with `TimedeltaIndex` or `Float64Index` created with pandas prior to version 0.20 (GH19939)
- Bug in `pandas.io.json.json_normalize()` where subrecords are not properly normalized if any subrecords values are `NoneType` (GH20030)
- Bug in `usecols` parameter in `read_csv()` where error is not raised correctly when passing a string. (GH20529)
- Bug in `HDFStore.keys()` when reading a file with a softlink causes exception (GH20523)
- Bug in `HDFStore.select_column()` where a key which is not a valid store raised an `AttributeError` instead of a `KeyError` (GH17912)

### 1.3.7.11 Plotting

- Better error message when attempting to plot but matplotlib is not installed (GH19810).
- `DataFrame.plot()` now raises a `ValueError` when the `x` or `y` argument is improperly formed (GH18671)
- Bug in `DataFrame.plot()` when `x` and `y` arguments given as positions caused incorrect referenced columns for line, bar and area plots (GH20056)
- Bug in formatting tick labels with `datetime.time()` and fractional seconds (GH18478).
- `Series.plot.kde()` has exposed the args `ind` and `bw_method` in the docstring (GH18461). The argument `ind` may now also be an integer (number of sample points).
- `DataFrame.plot()` now supports multiple columns to the `y` argument (GH19699)

### 1.3.7.12 Groupby/Resample/Rolling

- Bug when grouping by a single column and aggregating with a class like `list` or `tuple` (GH18079)
- Fixed regression in `DataFrame.groupby()` which would not emit an error when called with a tuple key not in the index (GH18798)
- Bug in `DataFrame.resample()` which silently ignored unsupported (or mistyped) options for `label`, `closed` and `convention` (GH19303)
- Bug in `DataFrame.groupby()` where tuples were interpreted as lists of keys rather than as keys (GH17979, GH18249)
- Bug in `DataFrame.groupby()` where aggregation by `first/last/min/max` was causing timestamps to lose precision (GH19526)
- Bug in `DataFrame.transform()` where particular aggregation functions were being incorrectly cast to match the dtype(s) of the grouped data (GH19200)
- Bug in `DataFrame.groupby()` passing the `on=` kwarg, and subsequently using `.apply()` (GH17813)
- Bug in `DataFrame.resample().aggregate` not raising a `KeyError` when aggregating a non-existent column (GH16766, GH19566)

- Bug in `DataFrameGroupBy.cumsum()` and `DataFrameGroupBy.cumprod()` when `skipna` was passed (GH19806)
- Bug in `DataFrame.resample()` that dropped timezone information (GH13238)
- Bug in `DataFrame.groupby()` where transformations using `np.all` and `np.any` were raising a `ValueError` (GH20653)
- Bug in `DataFrame.resample()` where `ffill`, `bfill`, `pad`, `backfill`, `fillna`, `interpolate`, and `asfreq` were ignoring `loffset`. (GH20744)
- Bug in `DataFrame.groupby()` when applying a function that has mixed data types and the user supplied function can fail on the grouping column (GH20949)
- Bug in `DataFrameGroupBy.rolling().apply()` where operations performed against the associated `DataFrameGroupBy` object could impact the inclusion of the grouped item(s) in the result (GH14013)

### 1.3.7.13 Sparse

- Bug in which creating a `SparseDataFrame` from a dense `Series` or an unsupported type raised an uncontrolled exception (GH19374)
- Bug in `SparseDataFrame.to_csv` causing exception (GH19384)
- Bug in `SparseSeries.memory_usage` which caused segfault by accessing non sparse elements (GH19368)
- Bug in constructing a `SparseArray`: if data is a scalar and index is defined it will coerce to `float64` regardless of scalar's dtype. (GH19163)

### 1.3.7.14 Reshaping

- Bug in `DataFrame.merge()` where referencing a `CategoricalIndex` by name, where the `by` kwarg would `KeyError` (GH20777)
- Bug in `DataFrame.stack()` which fails trying to sort mixed type levels under Python 3 (GH18310)
- Bug in `DataFrame.unstack()` which casts `int` to `float` if `columns` is a `MultiIndex` with unused levels (GH17845)
- Bug in `DataFrame.unstack()` which raises an error if index is a `MultiIndex` with unused labels on the unstacked level (GH18562)
- Fixed construction of a `Series` from a dict containing `NaN` as key (GH18480)
- Fixed construction of a `DataFrame` from a dict containing `NaN` as key (GH18455)
- Disabled construction of a `Series` where `len(index) > len(data) = 1`, which previously would broadcast the data item, and now raises a `ValueError` (GH18819)
- Suppressed error in the construction of a `DataFrame` from a dict containing scalar values when the corresponding keys are not included in the passed index (GH18600)
- Fixed (changed from object to `float64`) dtype of `DataFrame` initialized with axes, no data, and `dtype=int` (GH19646)
- Bug in `Series.rank()` where `Series` containing `NaT` modifies the `Series` inplace (GH18521)
- Bug in `cut()` which fails when using readonly arrays (GH18773)
- Bug in `DataFrame.pivot_table()` which fails when the `aggfunc` arg is of type string. The behavior is now consistent with other methods like `agg` and `apply` (GH18713)

- Bug in `DataFrame.merge()` in which merging using Index objects as vectors raised an Exception (GH19038)
- Bug in `DataFrame.stack()`, `DataFrame.unstack()`, `Series.unstack()` which were not returning subclasses (GH15563)
- Bug in timezone comparisons, manifesting as a conversion of the index to UTC in `.concat()` (GH18523)
- Bug in `concat()` when concatting sparse and dense series it returns only a `SparseDataFrame`. Should be a `DataFrame`. (GH18914, GH18686, and GH16874)
- Improved error message for `DataFrame.merge()` when there is no common merge key (GH19427)
- Bug in `DataFrame.join()` which does an outer instead of a left join when being called with multiple DataFrames and some have non-unique indices (GH19624)
- `Series.rename()` now accepts `axis` as a kwarg (GH18589)
- Bug in `rename()` where an Index of same-length tuples was converted to a MultiIndex (GH19497)
- Comparisons between `Series` and `Index` would return a `Series` with an incorrect name, ignoring the Index's name attribute (GH19582)
- Bug in `qcut()` where datetime and timedelta data with NaT present raised a `ValueError` (GH19768)
- Bug in `DataFrame.iterrows()`, which would infer strings not compliant to ISO8601 to datetimes (GH19671)
- Bug in `Series` constructor with `Categorical` where a `ValueError` is not raised when an index of different length is given (GH19342)
- Bug in `DataFrame.astype()` where column metadata is lost when converting to categorical or a dictionary of dtypes (GH19920)
- Bug in `cut()` and `qcut()` where timezone information was dropped (GH19872)
- Bug in `Series` constructor with a `dtype=str`, previously raised in some cases (GH19853)
- Bug in `get_dummies()`, and `select_dtypes()`, where duplicate column names caused incorrect behavior (GH20848)
- Bug in `isna()`, which cannot handle ambiguous typed lists (GH20675)
- Bug in `concat()` which raises an error when concatenating TZ-aware dataframes and all-NaT dataframes (GH12396)
- Bug in `concat()` which raises an error when concatenating empty TZ-aware series (GH18447)

#### 1.3.7.15 Other

- Improved error message when attempting to use a Python keyword as an identifier in a numexpr backed query (GH18221)
- Bug in accessing a `pandas.get_option()`, which raised `KeyError` rather than `OptionError` when looking up a non-existent option key in some cases (GH19789)
- Bug in `testing.assert_series_equal()` and `testing.assert_frame_equal()` for `Series` or `DataFrames` with differing unicode data (GH20503)

## 1.4 v0.22.0 (December 29, 2017)

This is a major release from 0.21.1 and includes a single, API-breaking change. We recommend that all users upgrade to this version after carefully reading the release note (singular!).

### 1.4.1 Backwards incompatible API changes

Pandas 0.22.0 changes the handling of empty and all-NA sums and products. The summary is that

- The sum of an empty or all-NA Series is now 0
- The product of an empty or all-NA Series is now 1
- We've added a `min_count` parameter to `.sum()` and `.prod()` controlling the minimum number of valid values for the result to be valid. If fewer than `min_count` non-NA values are present, the result is NA. The default is 0. To return NaN, the 0.21 behavior, use `min_count=1`.

Some background: In pandas 0.21, we fixed a long-standing inconsistency in the return value of all-NA series depending on whether or not bottleneck was installed. See [Sum/Prod of all-NA or empty Series/DataFrames is now consistently NaN](#). At the same time, we changed the sum and prod of an empty Series to also be NaN.

Based on feedback, we've partially reverted those changes.

#### 1.4.1.1 Arithmetic Operations

The default sum for empty or all-NA Series is now 0.

*pandas 0.21.x*

```
In [1]: pd.Series([]).sum()
Out[1]: nan

In [2]: pd.Series([np.nan]).sum()
Out[2]: nan
```

*pandas 0.22.0*

```
In [1]: pd.Series([]).sum()
Out[1]: 0.0

In [2]: pd.Series([np.nan]).sum()
Out[2]: 0.0
```

The default behavior is the same as pandas 0.20.3 with bottleneck installed. It also matches the behavior of NumPy's `np.nansum` on empty and all-NA arrays.

To have the sum of an empty series return NaN (the default behavior of pandas 0.20.3 without bottleneck, or pandas 0.21.x), use the `min_count` keyword.

```
In [3]: pd.Series([]).sum(min_count=1)
Out[3]: nan
```

Thanks to the `skipna` parameter, the `.sum` on an all-NA series is conceptually the same as the `.sum` of an empty one with `skipna=True` (the default).

```
In [4]: pd.Series([np.nan]).sum(min_count=1) # skipna=True by default
Out[4]: nan
```

The `min_count` parameter refers to the minimum number of *non-null* values required for a non-NA sum or product. `Series.prod()` has been updated to behave the same as `Series.sum()`, returning 1 instead.

```
In [5]: pd.Series([]).prod()
Out[5]: 1.0

In [6]: pd.Series([np.nan]).prod()
Out[6]: 1.0

In [7]: pd.Series([]).prod(min_count=1)
Out[7]: nan
```

These changes affect `DataFrame.sum()` and `DataFrame.prod()` as well. Finally, a few less obvious places in pandas are affected by this change.

#### 1.4.1.2 Grouping by a Categorical

Grouping by a `Categorical` and summing now returns 0 instead of NaN for categories with no observations. The product now returns 1 instead of NaN.

*pandas 0.21.x*

```
In [8]: grouper = pd.Categorical(['a', 'a'], categories=['a', 'b'])
In [9]: pd.Series([1, 2]).groupby(grouper).sum()
Out[9]:
a      3.0
b      NaN
dtype: float64
```

*pandas 0.22*

```
In [8]: grouper = pd.Categorical(['a', 'a'], categories=['a', 'b'])
In [9]: pd.Series([1, 2]).groupby(grouper).sum()
Out[9]:
a      3
b      0
dtype: int64
```

To restore the 0.21 behavior of returning NaN for unobserved groups, use `min_count>=1`.

```
In [10]: pd.Series([1, 2]).groupby(grouper).sum(min_count=1)
Out[10]:
a      3.0
b      NaN
dtype: float64
```

#### 1.4.1.3 Resample

The sum and product of all-NA bins has changed from NaN to 0 for sum and 1 for product.

*pandas 0.21.x*

```
In [11]: s = pd.Series([1, 1, np.nan, np.nan],
....:                  index=pd.date_range('2017', periods=4))
....: s
Out[11]:
2017-01-01    1.0
2017-01-02    1.0
2017-01-03    NaN
2017-01-04    NaN
Freq: D, dtype: float64

In [12]: s.resample('2d').sum()
Out[12]:
2017-01-01    2.0
2017-01-03    NaN
Freq: 2D, dtype: float64
```

*pandas 0.22.0*

```
In [11]: s = pd.Series([1, 1, np.nan, np.nan],
....:                  index=pd.date_range('2017', periods=4))
....:
In [12]: s.resample('2d').sum()
Out[12]:
2017-01-01    2.0
2017-01-03    0.0
dtype: float64
```

To restore the 0.21 behavior of returning NaN, use `min_count>=1`.

```
In [13]: s.resample('2d').sum(min_count=1)
Out[13]:
2017-01-01    2.0
2017-01-03    NaN
dtype: float64
```

In particular, upsampling and taking the sum or product is affected, as upsampling introduces missing values even if the original series was entirely valid.

*pandas 0.21.x*

```
In [14]: idx = pd.DatetimeIndex(['2017-01-01', '2017-01-02'])

In [15]: pd.Series([1, 2], index=idx).resample('12H').sum()
Out[15]:
2017-01-01 00:00:00    1.0
2017-01-01 12:00:00    NaN
2017-01-02 00:00:00    2.0
Freq: 12H, dtype: float64
```

*pandas 0.22.0*

```
In [14]: idx = pd.DatetimeIndex(['2017-01-01', '2017-01-02'])

In [15]: pd.Series([1, 2], index=idx).resample("12H").sum()
Out[15]:
2017-01-01 00:00:00    1
```

(continues on next page)



(continued from previous page)

```
2017-01-01 12:00:00    0
2017-01-02 00:00:00    2
Freq: 12H, dtype: int64
```

Once again, the `min_count` keyword is available to restore the 0.21 behavior.

```
In [16]: pd.Series([1, 2], index=idx).resample("12H").sum(min_count=1)
Out[16]:
2017-01-01 00:00:00    1.0
2017-01-01 12:00:00    NaN
2017-01-02 00:00:00    2.0
Freq: 12H, dtype: float64
```

#### 1.4.1.4 Rolling and Expanding

Rolling and expanding already have a `min_periods` keyword that behaves similar to `min_count`. The only case that changes is when doing a rolling or expanding sum with `min_periods=0`. Previously this returned `NaN`, when fewer than `min_periods` non-NA values were in the window. Now it returns 0.

*pandas 0.21.1*

```
In [17]: s = pd.Series([np.nan, np.nan])
In [18]: s.rolling(2, min_periods=0).sum()
Out[18]:
0    NaN
1    NaN
dtype: float64
```

*pandas 0.22.0*

```
In [17]: s = pd.Series([np.nan, np.nan])
In [18]: s.rolling(2, min_periods=0).sum()
Out[18]:
0    0.0
1    0.0
dtype: float64
```

The default behavior of `min_periods=None`, implying that `min_periods` equals the window size, is unchanged.

## 1.4.2 Compatibility

If you maintain a library that should work across pandas versions, it may be easiest to exclude pandas 0.21 from your requirements. Otherwise, all your `sum()` calls would need to check if the `Series` is empty before summing.

With `setuptools`, in your `setup.py` use:

```
install_requires=['pandas!=0.21.*', ...]
```

With `conda`, use

```
requirements:
  run:
    - pandas !=0.21.0,!=0.21.1
```

Note that the inconsistency in the return value for all-NA series is still there for pandas 0.20.3 and earlier. Avoiding pandas 0.21 will only help with the empty case.

## 1.5 v0.21.1 (December 12, 2017)

This is a minor bug-fix release in the 0.21.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Temporarily restore matplotlib datetime plotting functionality. This should resolve issues for users who implicitly relied on pandas to plot datetimes with matplotlib. See [here](#).
- Improvements to the Parquet IO functions introduced in 0.21.0. See [here](#).

### What's new in v0.21.1

- *Restore Matplotlib datetime Converter Registration*
- *New features*
  - *Improvements to the Parquet IO functionality*
  - *Other Enhancements*
- *Deprecations*
- *Performance Improvements*
- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Groupby/Resample/Rolling*
  - *Reshaping*
  - *Numeric*
  - *Categorical*
  - *String*

### 1.5.1 Restore Matplotlib datetime Converter Registration

Pandas implements some matplotlib converters for nicely formatting the axis labels on plots with `datetime` or `Period` values. Prior to pandas 0.21.0, these were implicitly registered with matplotlib, as a side effect of `import pandas`.

In pandas 0.21.0, we required users to explicitly register the converter. This caused problems for some users who relied on those converters being present for regular `matplotlib.pyplot` plotting methods, so we're temporarily reverting that change; pandas 0.21.1 again registers the converters on import, just like before 0.21.0.

We've added a new option to control the converters: `pd.options.plotting.matplotlib.register_converters`. By default, they are registered. Toggling this to `False` removes pandas' formatters and restore any converters we overwrote when registering them (GH18301).

We're working with the matplotlib developers to make this easier. We're trying to balance user convenience (automatically registering the converters) with import performance and best practices (importing pandas shouldn't have the side effect of overwriting any custom converters you've already set). In the future we hope to have most of the date-time formatting functionality in matplotlib, with just the pandas-specific converters in pandas. We'll then gracefully deprecate the automatic registration of converters in favor of users explicitly registering them when they want them.

## 1.5.2 New features

### 1.5.2.1 Improvements to the Parquet IO functionality

- `DataFrame.to_parquet()` will now write non-default indexes when the underlying engine supports it. The indexes will be preserved when reading back in with `read_parquet()` (GH18581).
- `read_parquet()` now allows to specify the columns to read from a parquet file (GH18154)
- `read_parquet()` now allows to specify kwargs which are passed to the respective engine (GH18216)

### 1.5.2.2 Other Enhancements

- `Timestamp.timestamp()` is now available in Python 2.7. (GH17329)
- `Grouper` and `TimeGrouper` now have a friendly repr output (GH18203).

## 1.5.3 Deprecations

- `pandas.tseries.register` has been renamed to `pandas.plotting.register_matplotlib_converters()` (GH18301)

## 1.5.4 Performance Improvements

- Improved performance of plotting large series/dataframes (GH18236).

## 1.5.5 Bug Fixes

### 1.5.5.1 Conversion

- Bug in `TimedeltaIndex` subtraction could incorrectly overflow when `NaT` is present (GH17791)
- Bug in `DatetimeIndex` subtracting datetimelike from `DatetimeIndex` could fail to overflow (GH18020)
- Bug in `IntervalIndex.copy()` when copying and `IntervalIndex` with non-default closed (GH18339)
- Bug in `DataFrame.to_dict()` where columns of datetime that are tz-aware were not converted to required arrays when used with `orient='records'`, raising `TypeError` (GH18372)
- Bug in `DatetimeIndex` and `date_range()` where mismatching tz-aware start and end timezones would not raise an err if `end.tzinfo` is `None` (GH18431)
- Bug in `Series.fillna()` which raised when passed a long integer on Python 2 (GH18159).

### 1.5.5.2 Indexing

- Bug in a boolean comparison of a `datetime.datetime` and a `datetime64[ns]` dtype Series (GH17965)
- Bug where a `MultiIndex` with more than a million records was not raising `AttributeError` when trying to access a missing attribute (GH18165)
- Bug in `IntervalIndex` constructor when a list of intervals is passed with non-default `closed` (GH18334)
- Bug in `Index.putmask` when an invalid mask passed (GH18368)
- Bug in masked assignment of a `timedelta64[ns]` dtype Series, incorrectly coerced to float (GH18493)

### 1.5.5.3 I/O

- Bug in class:~`pandas.io.stata.StataReader` not converting date/time columns with display formatting addressed (GH17990). Previously columns with display formatting were normally left as ordinal numbers and not converted to datetime objects.
- Bug in `read_csv()` when reading a compressed UTF-16 encoded file (GH18071)
- Bug in `read_csv()` for handling null values in index columns when specifying `na_filter=False` (GH5239)
- Bug in `read_csv()` when reading numeric category fields with high cardinality (GH18186)
- Bug in `DataFrame.to_csv()` when the table had `MultiIndex` columns, and a list of strings was passed in for header (GH5539)
- Bug in parsing integer datetime-like columns with specified format in `read_sql` (GH17855).
- Bug in `DataFrame.to_msgpack()` when serializing data of the `numpy.bool_` datatype (GH18390)
- Bug in `read_json()` not decoding when reading line delimited JSON from S3 (GH17200)
- Bug in `pandas.io.json.json_normalize()` to avoid modification of meta (GH18610)
- Bug in `to_latex()` where repeated multi-index values were not printed even though a higher level index differed from the previous row (GH14484)
- Bug when reading NaN-only categorical columns in `HDFStore` (GH18413)
- Bug in `DataFrame.to_latex()` with `longtable=True` where a latex multicolumn always spanned over three columns (GH17959)

### 1.5.5.4 Plotting

- Bug in `DataFrame.plot()` and `Series.plot()` with `DatetimeIndex` where a figure generated by them is not pickleable in Python 3 (GH18439)

### 1.5.5.5 Groupby/Resample/Rolling

- Bug in `DataFrame.resample(...).apply(...)` when there is a callable that returns different columns (GH15169)
- Bug in `DataFrame.resample(...)` when there is a time change (DST) and resampling frequency is 12h or higher (GH15549)
- Bug in `pd.DataFrameGroupBy.count()` when counting over a datetimelike column (GH13393)
- Bug in `rolling.var` where calculation is inaccurate with a zero-valued array (GH18430)

### 1.5.5.6 Reshaping

- Error message in `pd.merge_asof()` for key datatype mismatch now includes datatype of left and right key ([GH18068](#))
- Bug in `pd.concat` when empty and non-empty DataFrames or Series are concatenated ([GH18178](#) [GH18187](#))
- Bug in `DataFrame.filter(...)` when unicode is passed as a condition in Python 2 ([GH13101](#))
- Bug when merging empty DataFrames when `np.seterr(divide='raise')` is set ([GH17776](#))

### 1.5.5.7 Numeric

- Bug in `pd.Series.rolling.skew()` and `rolling.kurt()` with all equal values has floating issue ([GH18044](#))

### 1.5.5.8 Categorical

- Bug in `DataFrame.astype()` where casting to 'category' on an empty DataFrame causes a segmentation fault ([GH18004](#))
- Error messages in the testing module have been improved when items have different `CategoricalDtype` ([GH18069](#))
- `CategoricalIndex` can now correctly take a `pd.api.types.CategoricalDtype` as its dtype ([GH18116](#))
- Bug in `Categorical.unique()` returning read-only codes array when all categories were NaN ([GH18051](#))
- Bug in `DataFrame.groupby(axis=1)` with a `CategoricalIndex` ([GH18432](#))

### 1.5.5.9 String

- `Series.str.split()` will now propagate NaN values across all expanded columns instead of None ([GH18450](#))

## 1.6 v0.21.0 (October 27, 2017)

This is a major release from 0.20.3 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Integration with [Apache Parquet](#), including a new top-level `read_parquet()` function and `DataFrame.to_parquet()` method, see [here](#).
- New user-facing `pandas.api.types.CategoricalDtype` for specifying categoricals independent of the data, see [here](#).
- The behavior of `sum` and `prod` on all-NaN Series/DataFrames is now consistent and no longer depends on whether `bottleneck` is installed, and `sum` and `prod` on empty Series now return NaN instead of 0, see [here](#).
- Compatibility fixes for pypy, see [here](#).
- Additions to the `drop`, `reindex` and `rename` API to make them more consistent, see [here](#).

- Addition of the new methods `DataFrame.infer_objects` (see [here](#)) and `GroupBy.pipe` (see [here](#)).
- Indexing with a list of labels, where one or more of the labels is missing, is deprecated and will raise a `KeyError` in a future version, see [here](#).

Check the *API Changes* and *deprecations* before updating.

### What's new in v0.21.0

- *New features*
  - *Integration with Apache Parquet file format*
  - *infer\_objects type conversion*
  - *Improved warnings when attempting to create columns*
  - *drop now also accepts index/columns keywords*
  - *rename, reindex now also accept axis keyword*
  - *CategoricalDtype for specifying categoricals*
  - *GroupBy objects now have a pipe method*
  - *Categorical.rename\_categories accepts a dict-like*
  - *Other Enhancements*
- *Backwards incompatible API changes*
  - *Dependencies have increased minimum versions*
  - *Sum/Prod of all-NaN or empty Series/DataFrames is now consistently NaN*
  - *Indexing with a list with missing labels is Deprecated*
  - *NA naming Changes*
  - *Iteration of Series/Index will now return Python scalars*
  - *Indexing with a Boolean Index*
  - *PeriodIndex resampling*
  - *Improved error handling during item assignment in pd.eval*
  - *Dtype Conversions*
  - *MultiIndex Constructor with a Single Level*
  - *UTC Localization with Series*
  - *Consistency of Range Functions*
  - *No Automatic Matplotlib Converters*
  - *Other API Changes*
- *Deprecations*
  - *Series.select and DataFrame.select*
  - *Series.argmax and Series.argmin*
- *Removal of prior version deprecations/changes*
- *Performance Improvements*

- *Documentation Changes*
- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Groupby/Resample/Rolling*
  - *Sparse*
  - *Reshaping*
  - *Numeric*
  - *Categorical*
  - *PyPy*
  - *Other*

## 1.6.1 New features

### 1.6.1.1 Integration with Apache Parquet file format

Integration with [Apache Parquet](#), including a new top-level `read_parquet()` and `DataFrame.to_parquet()` method, see [here](#) (GH15838, GH17438).

[Apache Parquet](#) provides a cross-language, binary file format for reading and writing data frames efficiently. Parquet is designed to faithfully serialize and de-serialize `DataFrame`s, supporting all of the pandas dtypes, including extension dtypes such as datetime with timezones.

This functionality depends on either the [pyarrow](#) or [fastparquet](#) library. For more details, see [the IO docs on Parquet](#).

### 1.6.1.2 `infer_objects` type conversion

The `DataFrame.infer_objects()` and `Series.infer_objects()` methods have been added to perform dtype inference on object columns, replacing some of the functionality of the deprecated `convert_objects` method. See the documentation [here](#) for more details. (GH11221)

This method only performs soft conversions on object columns, converting Python objects to native types, but not any coercive conversions. For example:

```
In [1]: df = pd.DataFrame({'A': [1, 2, 3],
...:                      'B': np.array([1, 2, 3], dtype='object'),
...:                      'C': ['1', '2', '3']})
...:

In [2]: df.dtypes
Out[2]:
A      int64
B      object
C      object
```

(continues on next page)

(continued from previous page)

```
dtype: object

In [3]: df.infer_objects().dtypes
\\Out[3]:
A      int64
B      int64
C      object
dtype: object
```

Note that column 'C' was not converted - only scalar numeric types will be converted to a new type. Other types of conversion should be accomplished using the `to_numeric()` function (or `to_datetime()`, `to_timedelta()`).

```
In [4]: df = df.infer_objects()

In [5]: df['C'] = pd.to_numeric(df['C'], errors='coerce')

In [6]: df.dtypes
Out[6]:
A      int64
B      int64
C      int64
dtype: object
```

### 1.6.1.3 Improved warnings when attempting to create columns

New users are often puzzled by the relationship between column operations and attribute access on DataFrame instances (GH7175). One specific instance of this confusion is attempting to create a new column by setting an attribute on the DataFrame:

```
In[1]: df = pd.DataFrame({'one': [1., 2., 3.]})
In[2]: df.two = [4, 5, 6]
```

This does not raise any obvious exceptions, but also does not create a new column:

```
In[3]: df
Out[3]:
   one
0  1.0
1  2.0
2  3.0
```

Setting a list-like data structure into a new attribute now raises a `UserWarning` about the potential for unexpected behavior. See [Attribute Access](#).

### 1.6.1.4 drop now also accepts index/columns keywords

The `drop()` method has gained `index/columns` keywords as an alternative to specifying the axis. This is similar to the behavior of `reindex` (GH12392).

For example:





(continued from previous page)

```
In [15]: df.reindex([0, 1, 3], axis='index')
Out[15]:
```

	A	B
0	1.0	4.0
1	2.0	5.0
3	NaN	NaN

The “index, columns” style continues to work as before.

```
In [16]: df.rename(index=id, columns=str.lower)
Out[16]:
```

	a	b
4458898752	1	4
4458898784	2	5
4458898816	3	6

```
In [17]: df.reindex(index=[0, 1, 3], columns=['A', 'B', 'C'])
Out[17]:
```

	A	B	C
0	1.0	4.0	NaN
1	2.0	5.0	NaN
3	NaN	NaN	NaN

We *highly* encourage using named arguments to avoid confusion when using either style.

### 1.6.1.6 CategoricalDtype for specifying categoricals

`pandas.api.types.CategoricalDtype` has been added to the public API and expanded to include the categories and ordered attributes. A `CategoricalDtype` can be used to specify the set of categories and orderedness of an array, independent of the data. This can be useful for example, when converting string data to a `Categorical` ([GH14711](#), [GH15078](#), [GH16015](#), [GH17643](#)):

```
In [18]: from pandas.api.types import CategoricalDtype

In [19]: s = pd.Series(['a', 'b', 'c', 'a']) # strings

In [20]: dtype = CategoricalDtype(categories=['a', 'b', 'c', 'd'], ordered=True)

In [21]: s.astype(dtype)
Out[21]:
```

0	a
1	b
2	c
3	a

```
dtype: category
Categories (4, object): [a < b < c < d]
```

One place that deserves special mention is in `read_csv()`. Previously, with `dtype={'col': 'category'}`, the returned values and categories would always be strings.

```
In [22]: data = 'A,B\na,1\nb,2\nc,3'

In [23]: pd.read_csv(StringIO(data), dtype={'B': 'category'}).B.cat.categories
Out[23]: Index(['1', '2', '3'], dtype='object')
```

Notice the “object” dtype.

With a `CategoricalDtype` of all numerics, datetimes, or timedeltas, we can automatically convert to the correct type

```
In [24]: dtype = {'B': CategoricalDtype([1, 2, 3])}

In [25]: pd.read_csv(StringIO(data), dtype=dtype).B.cat.categories
Out[25]: Int64Index([1, 2, 3], dtype='int64')
```

The values have been correctly interpreted as integers.

The `.dtype` property of a `Categorical`, `CategoricalIndex` or a `Series` with categorical type will now return an instance of `CategoricalDtype`. While the repr has changed, `str(CategoricalDtype())` is still the string `'category'`. We'll take this moment to remind users that the *preferred* way to detect categorical data is to use `pandas.api.types.is_categorical_dtype()`, and not `str(dtype) == 'category'`.

See the [CategoricalDtype docs](#) for more.

### 1.6.1.7 GroupBy objects now have a pipe method

`GroupBy` objects now have a `pipe` method, similar to the one on `DataFrame` and `Series`, that allow for functions that take a `GroupBy` to be composed in a clean, readable syntax. (GH17871)

For a concrete example on combining `.groupby` and `.pipe`, imagine having a `DataFrame` with columns for stores, products, revenue and sold quantity. We'd like to do a groupwise calculation of *prices* (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable.

First we set the data:

```
In [26]: import numpy as np

In [27]: n = 1000

In [28]: df = pd.DataFrame({'Store': np.random.choice(['Store_1', 'Store_2'], n),
.....:                    'Product': np.random.choice(['Product_1', 'Product_2',
↪ 'Product_3'], n),
.....:                    'Revenue': (np.random.random(n)*50+10).round(2),
.....:                    'Quantity': np.random.randint(1, 10, size=n)})

In [29]: df.head(2)
Out[29]:
   Store  Product  Revenue  Quantity
0  Store_1  Product_3    54.28         3
1  Store_2  Product_2    30.91         1
```

Now, to find prices per store/product, we can simply do:

```
In [30]: (df.groupby(['Store', 'Product'])
.....:      .pipe(lambda grp: grp.Revenue.sum()/grp.Quantity.sum())
.....:      .unstack().round(2))
Out[30]:
Product  Product_1  Product_2  Product_3
Store
```

(continues on next page)

(continued from previous page)

Store_1	6.37	6.98	7.49
Store_2	7.60	7.01	7.13

See the *documentation* for more.

### 1.6.1.8 Categorical.rename\_categories accepts a dict-like

`rename_categories()` now accepts a dict-like argument for `new_categories`. The previous categories are looked up in the dictionary's keys and replaced if found. The behavior of missing and extra keys is the same as in `DataFrame.rename()`.

```
In [31]: c = pd.Categorical(['a', 'a', 'b'])
In [32]: c.rename_categories({"a": "eh", "b": "bee"})
Out[32]:
[eh, eh, bee]
Categories (2, object): [eh, bee]
```

**Warning:** To assist with upgrading pandas, `rename_categories` treats Series as list-like. Typically, Series are considered to be dict-like (e.g. in `.rename`, `.map`). In a future version of pandas `rename_categories` will change to treat them as dict-like. Follow the warning message's recommendations for writing future-proof code.

```
In [33]: c.rename_categories(pd.Series([0, 1], index=['a', 'c']))
FutureWarning: Treating Series 'new_categories' as a list-like and using the values.
In a future version, 'rename_categories' will treat Series like a dictionary.
For dict-like, use 'new_categories.to_dict()'
For list-like, use 'new_categories.values'.
Out[33]:
[0, 0, 1]
Categories (2, int64): [0, 1]
```

### 1.6.1.9 Other Enhancements

#### New functions or methods

- `nearest()` is added to support nearest-neighbor upsampling (GH17496).
- `Index` has added support for a `to_frame` method (GH15230).

#### New keywords

- Added a `skipna` parameter to `infer_dtype()` to support type inference in the presence of missing values (GH17059).
- `Series.to_dict()` and `DataFrame.to_dict()` now support an `into` keyword which allows you to specify the collections.Mapping subclass that you would like returned. The default is `dict`, which is backwards compatible. (GH16122)
- `Series.set_axis()` and `DataFrame.set_axis()` now support the `inplace` parameter. (GH14636)

- `Series.to_pickle()` and `DataFrame.to_pickle()` have gained a `protocol` parameter (GH16252). By default, this parameter is set to `HIGHEST_PROTOCOL`
- `read_feather()` has gained the `nthreads` parameter for multi-threaded operations (GH16359)
- `DataFrame.clip()` and `Series.clip()` have gained an `inplace` argument. (GH15388)
- `crosstab()` has gained a `margins_name` parameter to define the name of the row / column that will contain the totals when `margins=True`. (GH15972)
- `read_json()` now accepts a `chunksize` parameter that can be used when `lines=True`. If `chunksize` is passed, `read_json` now returns an iterator which reads in `chunksize` lines with each iteration. (GH17048)
- `read_json()` and `to_json()` now accept a `compression` argument which allows them to transparently handle compressed files. (GH17798)

## Various enhancements

- Improved the import time of pandas by about 2.25x. (GH16764)
- Support for PEP 519 – Adding a file system path protocol on most readers (e.g. `read_csv()`) and writers (e.g. `DataFrame.to_csv()`) (GH13823).
- Added a `__fspath__` method to `pd.HDFStore`, `pd.ExcelFile`, and `pd.ExcelWriter` to work properly with the file system path protocol (GH13823).
- The `validate` argument for `merge()` now checks whether a merge is one-to-one, one-to-many, many-to-one, or many-to-many. If a merge is found to not be an example of specified merge type, an exception of type `MergeError` will be raised. For more, see [here](#) (GH16270)
- Added support for PEP 518 (`pyproject.toml`) to the build system (GH16745)
- `RangeIndex.append()` now returns a `RangeIndex` object when possible (GH16212)
- `Series.rename_axis()` and `DataFrame.rename_axis()` with `inplace=True` now return `None` while renaming the axis inplace. (GH15704)
- `api.types.infer_dtype()` now infers decimals. (GH15690)
- `DataFrame.select_dtypes()` now accepts scalar values for include/exclude as well as list-like. (GH16855)
- `date_range()` now accepts ‘YS’ in addition to ‘AS’ as an alias for start of year. (GH9313)
- `date_range()` now accepts ‘Y’ in addition to ‘A’ as an alias for end of year. (GH9313)
- `DataFrame.add_prefix()` and `DataFrame.add_suffix()` now accept strings containing the ‘%’ character. (GH17151)
- Read/write methods that infer compression (`read_csv()`, `read_table()`, `read_pickle()`, and `to_pickle()`) can now infer from path-like objects, such as `pathlib.Path`. (GH17206)
- `read_sas()` now recognizes much more of the most frequently used date (datetime) formats in SAS7BDAT files. (GH15871)
- `DataFrame.items()` and `Series.items()` are now present in both Python 2 and 3 and is lazy in all cases. (GH13918, GH17213)
- `pandas.io.formats.style.Styler.where()` has been implemented as a convenience for `pandas.io.formats.style.Styler.applymap()`. (GH17474)
- `MultiIndex.is_monotonic_decreasing()` has been implemented. Previously returned `False` in all cases. (GH16554)

- `read_excel()` raises `ImportError` with a better message if `xlrd` is not installed. (GH17613)
- `DataFrame.assign()` will preserve the original order of `**kwargs` for Python 3.6+ users instead of sorting the column names. (GH14207)
- `Series.reindex()`, `DataFrame.reindex()`, `Index.get_indexer()` now support list-like argument for `tolerance`. (GH17367)

## 1.6.2 Backwards incompatible API changes

### 1.6.2.1 Dependencies have increased minimum versions

We have updated our minimum supported versions of dependencies (GH15206, GH15543, GH15214). If installed, we now require:

Package	Minimum Version	Required
Numpy	1.9.0	X
Matplotlib	1.4.3	
Scipy	0.14.0	
Bottleneck	1.0.0	

Additionally, support has been dropped for Python 3.4 (GH15251).

### 1.6.2.2 Sum/Prod of all-NaN or empty Series/DataFrames is now consistently NaN

---

**Note:** The changes described here have been partially reverted. See the [v0.22.0 Whatsnew](#) for more.

---

The behavior of `sum` and `prod` on all-NaN Series/DataFrames no longer depends on whether `bottleneck` is installed, and return value of `sum` and `prod` on an empty Series has changed (GH9422, GH15507).

Calling `sum` or `prod` on an empty or all-NaN Series, or columns of a DataFrame, will result in NaN. See the [docs](#).

```
In [33]: s = Series([np.nan])
```

Previously WITHOUT `bottleneck` installed:

```
In [2]: s.sum()
Out[2]: np.nan
```

Previously WITH `bottleneck`:

```
In [2]: s.sum()
Out[2]: 0.0
```

New Behavior, without regard to the `bottleneck` installation:

```
In [34]: s.sum()
Out[34]: 0.0
```

Note that this also changes the sum of an empty Series. Previously this always returned 0 regardless of a `bottleneck` installation:

```
In [1]: pd.Series([]).sum()
Out[1]: 0
```

but for consistency with the all-NaN case, this was changed to return NaN as well:

```
In [35]: pd.Series([]).sum()
Out[35]: 0.0
```

### 1.6.2.3 Indexing with a list with missing labels is Deprecated

Previously, selecting with a list of labels, where one or more labels were missing would always succeed, returning NaN for missing labels. This will now show a `FutureWarning`. In the future this will raise a `KeyError` ([GH15747](#)). This warning will trigger on a `DataFrame` or a `Series` for using `.loc[]` or `[]` when passing a list-of-labels with at least 1 missing label. See the [deprecation docs](#).

```
In [36]: s = pd.Series([1, 2, 3])

In [37]: s
Out[37]:
0      1
1      2
2      3
dtype: int64
```

#### Previous Behavior

```
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1      2.0
2      3.0
3      NaN
dtype: float64
```

#### Current Behavior

```
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc or [] with any missing label will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-
→listlike

Out[4]:
1      2.0
2      3.0
3      NaN
dtype: float64
```

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`

```
In [38]: s.reindex([1, 2, 3])
Out[38]:
1      2.0
2      3.0
```

(continues on next page)

(continued from previous page)

```
3      NaN
dtype: float64
```

Selection with all keys found is unchanged.

```
In [39]: s.loc[[1, 2]]
Out[39]:
1      2
2      3
dtype: int64
```

### 1.6.2.4 NA naming Changes

In order to promote more consistency among the pandas API, we have added additional top-level functions `isna()` and `notna()` that are aliases for `isnull()` and `notnull()`. The naming scheme is now more consistent with methods like `.dropna()` and `.fillna()`. Furthermore in all cases where `.isnull()` and `.notnull()` methods are defined, these have additional methods named `.isna()` and `.notna()`, these are included for classes `Categorical`, `Index`, `Series`, and `DataFrame`. (GH15001).

The configuration option `pd.options.mode.use_inf_as_null` is deprecated, and `pd.options.mode.use_inf_as_na` is added as a replacement.

### 1.6.2.5 Iteration of Series/Index will now return Python scalars

Previously, when using certain iteration methods for a `Series` with dtype `int` or `float`, you would receive a numpy scalar, e.g. a `np.int64`, rather than a Python `int`. Issue (GH10904) corrected this for `Series.tolist()` and `list(Series)`. This change makes all iteration methods consistent, in particular, for `__iter__()` and `.map()`; note that this only affects `int/float` dtypes. (GH13236, GH13258, GH14216).

```
In [40]: s = pd.Series([1, 2, 3])

In [41]: s
Out[41]:
0      1
1      2
2      3
dtype: int64
```

Previously:

```
In [2]: type(list(s)[0])
Out[2]: numpy.int64
```

New Behaviour:

```
In [42]: type(list(s)[0])
Out[42]: int
```

Furthermore this will now correctly box the results of iteration for `DataFrame.to_dict()` as well.

```
In [43]: d = {'a':[1], 'b':['b']}

In [44]: df = pd.DataFrame(d)
```



Previously:

```
In [8]: type(df.to_dict()['a'][0])
Out[8]: numpy.int64
```

New Behaviour:

```
In [45]: type(df.to_dict()['a'][0])
Out[45]: int
```

### 1.6.2.6 Indexing with a Boolean Index

Previously when passing a boolean Index to `.loc`, if the index of the Series/DataFrame had boolean labels, you would get a label based selection, potentially duplicating result labels, rather than a boolean indexing selection (where `True` selects elements), this was inconsistent how a boolean numpy array indexed. The new behavior is to act like a boolean numpy array indexer. ([GH17738](#))

Previous Behavior:

```
In [46]: s = pd.Series([1, 2, 3], index=[False, True, False])

In [47]: s
Out[47]:
False    1
True     2
False    3
dtype: int64
```

```
In [59]: s.loc[pd.Index([True, False, True])]
Out[59]:
True     2
False    1
False    3
True     2
dtype: int64
```

Current Behavior

```
In [48]: s.loc[pd.Index([True, False, True])]
Out[48]:
False    1
False    3
dtype: int64
```

Furthermore, previously if you had an index that was non-numeric (e.g. strings), then a boolean Index would raise a `KeyError`. This will now be treated as a boolean indexer.

Previously Behavior:

```
In [49]: s = pd.Series([1,2,3], index=['a', 'b', 'c'])

In [50]: s
Out[50]:
a     1
b     2
c     3
dtype: int64
```

```
In [39]: s.loc[pd.Index([True, False, True])]
KeyError: "None of [Index([True, False, True], dtype='object')] are in the [index]"
```

#### Current Behavior

```
In [51]: s.loc[pd.Index([True, False, True])]
Out[51]:
a      1
c      3
dtype: int64
```

### 1.6.2.7 PeriodIndex resampling

In previous versions of pandas, resampling a Series/DataFrame indexed by a PeriodIndex returned a DatetimeIndex in some cases (GH12884). Resampling to a multiplied frequency now returns a PeriodIndex (GH15944). As a minor enhancement, resampling a PeriodIndex can now handle NaT values (GH13224)

#### Previous Behavior:

```
In [1]: pi = pd.period_range('2017-01', periods=12, freq='M')
In [2]: s = pd.Series(np.arange(12), index=pi)
In [3]: resampled = s.resample('2Q').mean()
In [4]: resampled
Out[4]:
2017-03-31      1.0
2017-09-30      5.5
2018-03-31     10.0
Freq: 2Q-DEC, dtype: float64
In [5]: resampled.index
Out[5]: DatetimeIndex(['2017-03-31', '2017-09-30', '2018-03-31'], dtype=
↳ 'datetime64[ns]', freq='2Q-DEC')
```

#### New Behavior:

```
In [52]: pi = pd.period_range('2017-01', periods=12, freq='M')
In [53]: s = pd.Series(np.arange(12), index=pi)
In [54]: resampled = s.resample('2Q').mean()
In [55]: resampled
Out[55]:
2017Q1      2.5
2017Q3      8.5
Freq: 2Q-DEC, dtype: float64
In [56]: resampled.index
Out[56]:
PeriodIndex(['2017Q1', '2017Q3'], dtype='period[2Q-DEC]', freq='2Q-DEC')
```

Upsampling and calling `.ohlc()` previously returned a Series, basically identical to calling `.asfreq()`. OHLC upsampling now returns a DataFrame with columns open, high, low and close (GH13083). This is consistent with downsampling and DatetimeIndex behavior.

**New Behavior:**

#### 1.6.2.8 Improved error handling during item assignment in pd.eval

### 1.6. v0.21.0 (October 27, 2017)

```
In [61]: arr = np.array([1, 2, 3])
```

Previously, if you attempted the following expression, you would get a not very helpful error message:

```
In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)
...
IndexError: only integers, slices (':'), ellipsis ('...'), numpy.newaxis ('None')
and integer or boolean arrays are valid indices
```

This is a very long way of saying numpy arrays don't support string-item indexing. With this change, the error message is now this:

```
In [3]: pd.eval("a = 1 + 2", target=arr, inplace=True)
...
ValueError: Cannot assign expression output to target
```

It also used to be possible to evaluate expressions inplace, even if there was no item assignment:

```
In [4]: pd.eval("1 + 2", target=arr, inplace=True)
Out[4]: 3
```

However, this input does not make much sense because the output is not being assigned to the target. Now, a `ValueError` will be raised when such an input is passed in:

```
In [4]: pd.eval("1 + 2", target=arr, inplace=True)
...
ValueError: Cannot operate inplace if there is no assignment
```

### 1.6.2.9 Dtype Conversions

Previously assignments, `.where()` and `.fillna()` with a `bool` assignment, would coerce to same the type (e.g. `int` / `float`), or raise for datetimelikes. These will now preserve the bools with `object` dtypes. ([GH16821](#)).

```
In [62]: s = Series([1, 2, 3])
```

```
In [5]: s[1] = True

In [6]: s
Out[6]:
0    1
1    1
2    3
dtype: int64
```

New Behavior

```
In [63]: s[1] = True

In [64]: s
Out[64]:
0    1
1    True
2    3
dtype: object
```

Previously, as assignment to a datetimelike with a non-datetimelike would coerce the non-datetime-like item being assigned ([GH14145](#)).

```
In [65]: s = pd.Series([pd.Timestamp('2011-01-01'), pd.Timestamp('2012-01-01')])
```

```
In [1]: s[1] = 1

In [2]: s
Out[2]:
0    2011-01-01 00:00:00.000000000
1    1970-01-01 00:00:00.000000001
dtype: datetime64[ns]
```

These now coerce to object dtype.

```
In [66]: s[1] = 1

In [67]: s
Out[67]:
0    2011-01-01 00:00:00
1
dtype: object
```

- Inconsistent behavior in `.where()` with datetimelikes which would raise rather than coerce to object ([GH16402](#))
- Bug in assignment against `int64` data with `np.ndarray` with `float64` dtype may keep `int64` dtype ([GH14001](#))

### 1.6.2.10 MultiIndex Constructor with a Single Level

The `MultiIndex` constructors no longer squeezes a `MultiIndex` with all length-one levels down to a regular `Index`. This affects all the `MultiIndex` constructors. ([GH17178](#))

Previous behavior:

```
In [2]: pd.MultiIndex.from_tuples([('a',), ('b',)])
Out[2]: Index(['a', 'b'], dtype='object')
```

Length 1 levels are no longer special-cased. They behave exactly as if you had length 2+ levels, so a `MultiIndex` is always returned from all of the `MultiIndex` constructors:

```
In [68]: pd.MultiIndex.from_tuples([('a',), ('b',)])
Out[68]:
MultiIndex(levels=[['a', 'b']],
            labels=[[0, 1]])
```

### 1.6.2.11 UTC Localization with Series

Previously, `to_datetime()` did not localize datetime Series data when `utc=True` was passed. Now, `to_datetime()` will correctly localize Series with a `datetime64[ns, UTC]` dtype to be consistent with how list-like and `Index` data are handled. ([GH6415](#)).

Previous Behavior

```
In [69]: s = Series(['20130101 00:00:00'] * 3)
```

```
In [12]: pd.to_datetime(s, utc=True)
```

```
Out [12]:
```

```
0    2013-01-01
1    2013-01-01
2    2013-01-01
dtype: datetime64[ns]
```

### New Behavior

```
In [70]: pd.to_datetime(s, utc=True)
```

```
Out [70]:
```

```
0    2013-01-01 00:00:00+00:00
1    2013-01-01 00:00:00+00:00
2    2013-01-01 00:00:00+00:00
dtype: datetime64[ns, UTC]
```

Additionally, DataFrames with datetime columns that were parsed by `read_sql_table()` and `read_sql_query()` will also be localized to UTC only if the original SQL columns were timezone aware datetime columns.

## 1.6.2.12 Consistency of Range Functions

In previous versions, there were some inconsistencies between the various range functions: `date_range()`, `bdate_range()`, `period_range()`, `timedelta_range()`, and `interval_range()`. (GH17471).

One of the inconsistent behaviors occurred when the start, end and period parameters were all specified, potentially leading to ambiguous ranges. When all three parameters were passed, `interval_range` ignored the period parameter, `period_range` ignored the end parameter, and the other range functions raised. To promote consistency among the range functions, and avoid potentially ambiguous ranges, `interval_range` and `period_range` will now raise when all three parameters are passed.

### Previous Behavior:

```
In [2]: pd.interval_range(start=0, end=4, periods=6)
```

```
Out [2]:
```

```
IntervalIndex([(0, 1], (1, 2], (2, 3]]
              closed='right',
              dtype='interval[int64]')
```

```
In [3]: pd.period_range(start='2017Q1', end='2017Q4', periods=6, freq='Q')
```

```
Out [3]: PeriodIndex(['2017Q1', '2017Q2', '2017Q3', '2017Q4', '2018Q1', '2018Q2'],
                    dtype='period[Q-DEC]', freq='Q-DEC')
```

### New Behavior:

```
In [2]: pd.interval_range(start=0, end=4, periods=6)
```

```
-----
ValueError: Of the three parameters: start, end, and periods, exactly two must be_
→ specified
```

```
In [3]: pd.period_range(start='2017Q1', end='2017Q4', periods=6, freq='Q')
```

```
-----
ValueError: Of the three parameters: start, end, and periods, exactly two must be_
→ specified
```

Additionally, the endpoint parameter `end` was not included in the intervals produced by `interval_range`. However, all other range functions include `end` in their output. To promote consistency among the range functions, `interval_range` will now include `end` as the right endpoint of the final interval, except if `freq` is specified in a way which skips `end`.

Previous Behavior:

```
In [4]: pd.interval_range(start=0, end=4)
Out[4]:
IntervalIndex([(0, 1], (1, 2], (2, 3]]
              closed='right',
              dtype='interval[int64]')
```

New Behavior:

```
In [71]: pd.interval_range(start=0, end=4)
Out[71]:
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4]]
              closed='right',
              dtype='interval[int64]')
```

### 1.6.2.13 No Automatic Matplotlib Converters

Pandas no longer registers our date, time, datetime, datetime64, and Period converters with matplotlib when pandas is imported. Matplotlib plot methods (`plt.plot`, `ax.plot`, ...), will not nicely format the x-axis for `DatetimeIndex` or `PeriodIndex` values. You must explicitly register these methods:

Pandas built-in `Series.plot` and `DataFrame.plot` *will* register these converters on first-use ([GH17710](#)).

---

**Note:** This change has been temporarily reverted in pandas 0.21.1, for more details see [here](#).

---

### 1.6.2.14 Other API Changes

- The Categorical constructor no longer accepts a scalar for the `categories` keyword. ([GH16022](#))
- Accessing a non-existent attribute on a closed `HDFStore` will now raise an `AttributeError` rather than a `ClosedFileError` ([GH16301](#))
- `read_csv()` now issues a `UserWarning` if the `names` parameter contains duplicates ([GH17095](#))
- `read_csv()` now treats `'null'` and `'n/a'` strings as missing values by default ([GH16471](#), [GH16078](#))
- `pandas.HDFStore`'s string representation is now faster and less detailed. For the previous behavior, use `pandas.HDFStore.info()`. ([GH16503](#)).
- Compression defaults in HDF stores now follow pytables standards. Default is no compression and if `complib` is missing and `complevel > 0` `zlib` is used ([GH15943](#))
- `Index.get_indexer_non_unique()` now returns a `ndarray` indexer rather than an `Index`; this is consistent with `Index.get_indexer()` ([GH16819](#))
- Removed the `@slow` decorator from `pandas.util.testing`, which caused issues for some downstream packages' test suites. Use `@pytest.mark.slow` instead, which achieves the same thing ([GH16850](#))
- Moved definition of `MergeError` to the `pandas.errors` module.

- The signature of `Series.set_axis()` and `DataFrame.set_axis()` has been changed from `set_axis(axis, labels)` to `set_axis(labels, axis=0)`, for consistency with the rest of the API. The old signature is deprecated and will show a `FutureWarning` (GH14636)
- `Series.argmax()` and `Series.argmin()` will now raise a `TypeError` when used with object dtypes, instead of a `ValueError` (GH13595)
- `Period` is now immutable, and will now raise an `AttributeError` when a user tries to assign a new value to the ordinal or freq attributes (GH17116).
- `to_datetime()` when passed a tz-aware `origin=` kwarg will now raise a more informative `ValueError` rather than a `TypeError` (GH16842)
- `to_datetime()` now raises a `ValueError` when format includes `%W` or `%U` without also including day of the week and calendar year (GH16774)
- Renamed non-functional index to `index_col` in `read_stata()` to improve API consistency (GH16342)
- Bug in `DataFrame.drop()` caused boolean labels `False` and `True` to be treated as labels 0 and 1 respectively when dropping indices from a numeric index. This will now raise a `ValueError` (GH16877)
- Restricted `DateOffset` keyword arguments. Previously, `DateOffset` subclasses allowed arbitrary keyword arguments which could lead to unexpected behavior. Now, only valid arguments will be accepted. (GH17176).

### 1.6.3 Deprecations

- `DataFrame.from_csv()` and `Series.from_csv()` have been deprecated in favor of `read_csv()` (GH4191)
- `read_excel()` has deprecated `sheetname` in favor of `sheet_name` for consistency with `.to_excel()` (GH10559).
- `read_excel()` has deprecated `parse_cols` in favor of `usecols` for consistency with `read_csv()` (GH4988)
- `read_csv()` has deprecated the `tupleize_cols` argument. Column tuples will always be converted to a `MultiIndex` (GH17060)
- `DataFrame.to_csv()` has deprecated the `tupleize_cols` argument. Multi-index columns will be always written as rows in the CSV file (GH17060)
- The `convert` parameter has been deprecated in the `.take()` method, as it was not being respected (GH16948)
- `pd.options.html.border` has been deprecated in favor of `pd.options.display.html.border` (GH15793).
- `SeriesGroupBy.nth()` has deprecated `True` in favor of `'all'` for its kwarg `dropna` (GH11038).
- `DataFrame.as_blocks()` is deprecated, as this is exposing the internal implementation (GH17302)
- `pd.TimeGrouper` is deprecated in favor of `pandas.Grouper` (GH16747)
- `cdate_range` has been deprecated in favor of `bdate_range()`, which has gained `weekmask` and `holidays` parameters for building custom frequency date ranges. See the [documentation](#) for more details (GH17596)
- passing categories or ordered kwargs to `Series.astype()` is deprecated, in favor of passing a `CategoricalDtype` (GH17636)
- `.get_value` and `.set_value` on `Series`, `DataFrame`, `Panel`, `SparseSeries`, and `SparseDataFrame` are deprecated in favor of using `.iat[]` or `.at[]` accessors (GH15269)



- Passing a non-existent column in `.to_excel(..., columns=)` is deprecated and will raise a `KeyError` in the future (GH17295)
- `raise_on_error` parameter to `Series.where()`, `Series.mask()`, `DataFrame.where()`, `DataFrame.mask()` is deprecated, in favor of `errors=` (GH14968)
- Using `DataFrame.rename_axis()` and `Series.rename_axis()` to alter index or column *labels* is now deprecated in favor of using `.rename`. `rename_axis` may still be used to alter the name of the index or columns (GH17833).
- `reindex_axis()` has been deprecated in favor of `reindex()`. See [here](#) for more (GH17833).

### 1.6.3.1 Series.select and DataFrame.select

The `Series.select()` and `DataFrame.select()` methods are deprecated in favor of using `df.loc[labels.map(crit)]` (GH12401)

```
In [72]: df = DataFrame({'A': [1, 2, 3]}, index=['foo', 'bar', 'baz'])
```

```
In [3]: df.select(lambda x: x in ['bar', 'baz'])
```

FutureWarning: select is deprecated and will be removed in a future release. You can use `.loc[crit]` as a replacement

```
Out [3]:
```

```
      A
bar   2
baz   3
```

```
In [73]: df.loc[df.index.map(lambda x: x in ['bar', 'baz'])]
```

```
Out [73]:
```

```
      A
bar   2
baz   3
```

### 1.6.3.2 Series.argmax and Series.argmin

The behavior of `Series.argmax()` and `Series.argmin()` have been deprecated in favor of `Series.idxmax()` and `Series.idxmin()`, respectively (GH16830).

For compatibility with NumPy arrays, `pd.Series` implements `argmax` and `argmin`. Since pandas 0.13.0, `argmax` has been an alias for `pandas.Series.idxmax()`, and `argmin` has been an alias for `pandas.Series.idxmin()`. They return the *label* of the maximum or minimum, rather than the *position*.

We've deprecated the current behavior of `Series.argmax` and `Series.argmin`. Using either of these will emit a `FutureWarning`. Use `Series.idxmax()` if you want the label of the maximum. Use `Series.values.argmax()` if you want the position of the maximum. Likewise for the minimum. In a future release `Series.argmax` and `Series.argmin` will return the position of the maximum or minimum.

## 1.6.4 Removal of prior version deprecations/changes

- `read_excel()` has dropped the `has_index_names` parameter (GH10967)
- The `pd.options.display.height` configuration has been dropped (GH3663)
- The `pd.options.display.line_width` configuration has been dropped (GH2881)
- The `pd.options.display.mpl_style` configuration has been dropped (GH12190)

- Index has dropped the `.sym_diff()` method in favor of `.symmetric_difference()` (GH12591)
- Categorical has dropped the `.order()` and `.sort()` methods in favor of `.sort_values()` (GH12882)
- `eval()` and `DataFrame.eval()` have changed the default of `inplace` from `None` to `False` (GH11149)
- The function `get_offset_name` has been dropped in favor of the `.freqstr` attribute for an offset (GH11834)
- pandas no longer tests for compatibility with hdf5-files created with pandas < 0.11 (GH17404).

## 1.6.5 Performance Improvements

- Improved performance of instantiating `SparseDataFrame` (GH16773)
- `Series.dt` no longer performs frequency inference, yielding a large speedup when accessing the attribute (GH17210)
- Improved performance of `set_categories()` by not materializing the values (GH17508)
- `Timestamp.microsecond` no longer re-computes on attribute access (GH17331)
- Improved performance of the `CategoricalIndex` for data that is already categorical dtype (GH17513)
- Improved performance of `RangeIndex.min()` and `RangeIndex.max()` by using `RangeIndex` properties to perform the computations (GH17607)

## 1.6.6 Documentation Changes

- Several NaT method docstrings (e.g. `NaT.ctime()`) were incorrect (GH17327)
- The documentation has had references to versions < v0.17 removed and cleaned up (GH17442, GH17442, GH17404 & GH17504)

## 1.6.7 Bug Fixes

### 1.6.7.1 Conversion

- Bug in assignment against datetime-like data with `int` may incorrectly convert to datetime-like (GH14145)
- Bug in assignment against `int64` data with `np.ndarray` with `float64` dtype may keep `int64` dtype (GH14001)
- Fixed the return type of `IntervalIndex.is_non_overlapping_monotonic` to be a Python `bool` for consistency with similar attributes/methods. Previously returned a `numpy.bool_`. (GH17237)
- Bug in `IntervalIndex.is_non_overlapping_monotonic` when intervals are closed on both sides and overlap at a point (GH16560)
- Bug in `Series.fillna()` returns frame when `inplace=True` and value is dict (GH16156)
- Bug in `Timestamp.weekday_name` returning a UTC-based weekday name when localized to a timezone (GH17354)
- Bug in `Timestamp.replace` when replacing `tzinfo` around DST changes (GH15683)
- Bug in `Timedelta` construction and arithmetic that would not propagate the `Overflow` exception (GH17367)

- Bug in `astype()` converting to object dtype when passed extension type classes (`DatetimeTZDtype`, `CategoricalDtype`) rather than instances. Now a `TypeError` is raised when a class is passed ([GH17780](#)).
- Bug in `to_numeric()` in which elements were not always being coerced to numeric when `errors='coerce'` ([GH17007](#), [GH17125](#))
- Bug in `DataFrame` and `Series` constructors where range objects are converted to `int32` dtype on Windows instead of `int64` ([GH16804](#))

### 1.6.7.2 Indexing

- When called with a null slice (e.g. `df.iloc[:]`), the `.iloc` and `.loc` indexers return a shallow copy of the original object. Previously they returned the original object. ([GH13873](#)).
- When called on an unsorted `MultiIndex`, the `loc` indexer now will raise `UnsortedIndexError` only if proper slicing is used on non-sorted levels ([GH16734](#)).
- Fixes regression in 0.20.3 when indexing with a string on a `TimedeltaIndex` ([GH16896](#)).
- Fixed `TimedeltaIndex.get_loc()` handling of `np.timedelta64` inputs ([GH16909](#)).
- Fix `MultiIndex.sort_index()` ordering when ascending argument is a list, but not all levels are specified, or are in a different order ([GH16934](#)).
- Fixes bug where indexing with `np.inf` caused an `OverflowError` to be raised ([GH16957](#))
- Bug in reindexing on an empty `CategoricalIndex` ([GH16770](#))
- Fixes `DataFrame.loc` for setting with alignment and tz-aware `DatetimeIndex` ([GH16889](#))
- Avoids `IndexError` when passing an `Index` or `Series` to `.iloc` with older numpy ([GH17193](#))
- Allow unicode empty strings as placeholders in multilevel columns in Python 2 ([GH17099](#))
- Bug in `.iloc` when used with inplace addition or assignment and an int indexer on a `MultiIndex` causing the wrong indexes to be read from and written to ([GH17148](#))
- Bug in `.isin()` in which checking membership in empty `Series` objects raised an error ([GH16991](#))
- Bug in `CategoricalIndex` reindexing in which specified indices containing duplicates were not being respected ([GH17323](#))
- Bug in intersection of `RangeIndex` with negative step ([GH17296](#))
- Bug in `IntervalIndex` where performing a scalar lookup fails for included right endpoints of non-overlapping monotonic decreasing indexes ([GH16417](#), [GH17271](#))
- Bug in `DataFrame.first_valid_index()` and `DataFrame.last_valid_index()` when no valid entry ([GH17400](#))
- Bug in `Series.rename()` when called with a callable, incorrectly alters the name of the `Series`, rather than the name of the `Index`. ([GH17407](#))
- Bug in `String.str_get()` raises `IndexError` instead of inserting NaNs when using a negative index. ([GH17704](#))

### 1.6.7.3 I/O

- Bug in `read_hdf()` when reading a timezone aware index from fixed format `HDFStore` ([GH17618](#))
- Bug in `read_csv()` in which columns were not being thoroughly de-duplicated ([GH17060](#))
- Bug in `read_csv()` in which specified column names were not being thoroughly de-duplicated ([GH17095](#))

- Bug in `read_csv()` in which non integer values for the header argument generated an unhelpful / unrelated error message (GH16338)
- Bug in `read_csv()` in which memory management issues in exception handling, under certain conditions, would cause the interpreter to segfault (GH14696, GH16798).
- Bug in `read_csv()` when called with `low_memory=False` in which a CSV with at least one column > 2GB in size would incorrectly raise a `MemoryError` (GH16798).
- Bug in `read_csv()` when called with a single-element list header would return a `DataFrame` of all NaN values (GH7757)
- Bug in `DataFrame.to_csv()` defaulting to 'ascii' encoding in Python 3, instead of 'utf-8' (GH17097)
- Bug in `read_stata()` where value labels could not be read when using an iterator (GH16923)
- Bug in `read_stata()` where the index was not set (GH16342)
- Bug in `read_html()` where import check fails when run in multiple threads (GH16928)
- Bug in `read_csv()` where automatic delimiter detection caused a `TypeError` to be thrown when a bad line was encountered rather than the correct error message (GH13374)
- Bug in `DataFrame.to_html()` with `notebook=True` where `DataFrames` with named indices or non-`MultiIndex` indices had undesired horizontal or vertical alignment for column or row labels, respectively (GH16792)
- Bug in `DataFrame.to_html()` in which there was no validation of the `justify` parameter (GH17527)
- Bug in `HDFStore.select()` when reading a contiguous mixed-data table featuring `VArray` (GH17021)
- Bug in `to_json()` where several conditions (including objects with unprintable symbols, objects with deep recursion, overlong labels) caused segfaults instead of raising the appropriate exception (GH14256)

#### 1.6.7.4 Plotting

- Bug in plotting methods using `secondary_y` and `fontsize` not setting secondary axis font size (GH12565)
- Bug when plotting `timedelta` and `datetime` dtypes on y-axis (GH16953)
- Line plots no longer assume monotonic x data when calculating xlimits, they show the entire lines now even for unsorted x data. (GH11310, GH11471)
- With matplotlib 2.0.0 and above, calculation of x limits for line plots is left to matplotlib, so that its new default settings are applied. (GH15495)
- Bug in `Series.plot.bar` or `DataFrame.plot.bar` with `y` not respecting user-passed `color` (GH16822)
- Bug causing `plotting.parallel_coordinates` to reset the random seed when using random colors (GH17525)

#### 1.6.7.5 Groupby/Resample/Rolling

- Bug in `DataFrame.resample(...).size()` where an empty `DataFrame` did not return a `Series` (GH14962)
- Bug in `infer_freq()` causing indices with 2-day gaps during the working week to be wrongly inferred as business daily (GH16624)
- Bug in `.rolling(...).quantile()` which incorrectly used different defaults than `Series.quantile()` and `DataFrame.quantile()` (GH9413, GH16211)

- Bug in `groupby.transform()` that would coerce boolean dtypes back to float ([GH16875](#))
- Bug in `Series.resample(...).apply()` where an empty Series modified the source index and did not return the name of a Series ([GH14313](#))
- Bug in `.rolling(...).apply(...)` with a DataFrame with a DatetimeIndex, a window of a timedelta-convertible and `min_periods >= 1` ([GH15305](#))
- Bug in `DataFrame.groupby` where index and column keys were not recognized correctly when the number of keys equaled the number of elements on the groupby axis ([GH16859](#))
- Bug in `groupby.nunique()` with TimeGrouper which cannot handle NaT correctly ([GH17575](#))
- Bug in `DataFrame.groupby` where a single level selection from a MultiIndex unexpectedly sorts ([GH17537](#))
- Bug in `DataFrame.groupby` where spurious warning is raised when Grouper object is used to override ambiguous column name ([GH17383](#))
- Bug in TimeGrouper differs when passes as a list and as a scalar ([GH17530](#))

#### 1.6.7.6 Sparse

- Bug in `SparseSeries` raises `AttributeError` when a dictionary is passed in as data ([GH16905](#))
- Bug in `SparseDataFrame.fillna()` not filling all NaNs when frame was instantiated from SciPy sparse matrix ([GH16112](#))
- Bug in `SparseSeries.unstack()` and `SparseDataFrame.stack()` ([GH16614](#), [GH15045](#))
- Bug in `make_sparse()` treating two numeric/boolean data, which have same bits, as same when array dtype is object ([GH17574](#))
- `SparseArray.all()` and `SparseArray.any()` are now implemented to handle `SparseArray`, these were used but not implemented ([GH17570](#))

#### 1.6.7.7 Reshaping

- Joining/Merging with a non unique PeriodIndex raised a `TypeError` ([GH16871](#))
- Bug in `crosstab()` where non-aligned series of integers were casted to float ([GH17005](#))
- Bug in merging with categorical dtypes with datetimelikes incorrectly raised a `TypeError` ([GH16900](#))
- Bug when using `isin()` on a large object series and large comparison array ([GH16012](#))
- Fixes regression from 0.20, `Series.aggregate()` and `DataFrame.aggregate()` allow dictionaries as return values again ([GH16741](#))
- Fixes dtype of result with integer dtype input, from `pivot_table()` when called with `margins=True` ([GH17013](#))
- Bug in `crosstab()` where passing two Series with the same name raised a `KeyError` ([GH13279](#))
- `Series.argmax()`, `Series.argmin()`, and their counterparts on DataFrame and groupby objects work correctly with floating point data that contains infinite values ([GH13595](#)).
- Bug in `unique()` where checking a tuple of strings raised a `TypeError` ([GH17108](#))
- Bug in `concat()` where order of result index was unpredictable if it contained non-comparable elements ([GH17344](#))

- Fixes regression when sorting by multiple columns on a `datetime64` dtype Series with NaT values (GH16836)
- Bug in `pivot_table()` where the result's columns did not preserve the categorical dtype of columns when `dropna` was `False` (GH17842)
- Bug in `DataFrame.drop_duplicates` where dropping with non-unique column names raised a `ValueError` (GH17836)
- Bug in `unstack()` which, when called on a list of levels, would discard the `fillna` argument (GH13971)
- Bug in the alignment of range objects and other list-likes with `DataFrame` leading to operations being performed row-wise instead of column-wise (GH17901)

#### 1.6.7.8 Numeric

- Bug in `.clip()` with `axis=1` and a list-like for `threshold` is passed; previously this raised `ValueError` (GH15390)
- `Series.clip()` and `DataFrame.clip()` now treat NA values for upper and lower arguments as `None` instead of raising `ValueError` (GH17276).

#### 1.6.7.9 Categorical

- Bug in `Series.isin()` when called with a categorical (GH16639)
- Bug in the categorical constructor with empty values and categories causing the `.categories` to be an empty `Float64Index` rather than an empty `Index` with object dtype (GH17248)
- Bug in categorical operations with `Series.cat` not preserving the original Series' name (GH17509)
- Bug in `DataFrame.merge()` failing for categorical columns with boolean/int data types (GH17187)
- Bug in constructing a `Categorical/CategoricalDtype` when the specified categories are of categorical type (GH17884).

#### 1.6.7.10 PyPy

- Compatibility with PyPy in `read_csv()` with `usecols=[<unsorted ints>]` and `read_json()` (GH17351)
- Split tests into cases for CPython and PyPy where needed, which highlights the fragility of index matching with `float('nan')`, `np.nan` and `NAT` (GH17351)
- Fix `DataFrame.memory_usage()` to support PyPy. Objects on PyPy do not have a fixed size, so an approximation is used instead (GH17228)

#### 1.6.7.11 Other

- Bug where some inplace operators were not being wrapped and produced a copy when invoked (GH12962)
- Bug in `eval()` where the `inplace` parameter was being incorrectly handled (GH16732)

## 1.7 v0.20.3 (July 7, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small regression fixes and bug fixes. We recommend that all users upgrade to this version.

### What's new in v0.20.3

- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Reshaping*
  - *Categorical*

### 1.7.1 Bug Fixes

- Fixed a bug in failing to compute rolling computations of a column-MultiIndexed DataFrame ([GH16789](#), [GH16825](#))
- Fixed a pytest marker failing downstream packages' tests suites ([GH16680](#))

#### 1.7.1.1 Conversion

- Bug in pickle compat prior to the v0.20.x series, when UTC is a timezone in a Series/DataFrame/Index ([GH16608](#))
- Bug in Series construction when passing a Series with dtype='category' ([GH16524](#)).
- Bug in `DataFrame.astype()` when passing a Series as the dtype kwarg. ([GH16717](#)).

#### 1.7.1.2 Indexing

- Bug in `Float64Index` causing an empty array instead of None to be returned from `.get(np.nan)` on a Series whose index did not contain any NaN s ([GH8569](#))
- Bug in `MultiIndex.isin` causing an error when passing an empty iterable ([GH16777](#))
- Fixed a bug in a slicing DataFrame/Series that have a `TimedeltaIndex` ([GH16637](#))

#### 1.7.1.3 I/O

- Bug in `read_csv()` in which files weren't opened as binary files by the C engine on Windows, causing EOF characters mid-field, which would fail ([GH16039](#), [GH16559](#), [GH16675](#))
- Bug in `read_hdf()` in which reading a Series saved to an HDF file in 'fixed' format fails when an explicit `mode='r'` argument is supplied ([GH16583](#))

- Bug in `DataFrame.to_latex()` where `bold_rows` was wrongly specified to be `True` by default, whereas in reality row labels remained non-bold whatever parameter provided. (GH16707)
- Fixed an issue with `DataFrame.style()` where generated element ids were not unique (GH16780)
- Fixed loading a `DataFrame` with a `PeriodIndex`, from a `format='fixed'` `HDFStore`, in Python 3, that was written in Python 2 (GH16781)

#### 1.7.1.4 Plotting

- Fixed regression that prevented RGB and RGBA tuples from being used as color arguments (GH16233)
- Fixed an issue with `DataFrame.plot.scatter()` that incorrectly raised a `KeyError` when categorical data is used for plotting (GH16199)

#### 1.7.1.5 Reshaping

- `PeriodIndex / TimedeltaIndex.join` was missing the `sort=` kwarg (GH16541)
- Bug in joining on a `MultiIndex` with a `category` dtype for a level (GH16627).
- Bug in `merge()` when merging/joining with multiple categorical columns (GH16767)

#### 1.7.1.6 Categorical

- Bug in `DataFrame.sort_values` not respecting the `kind` parameter with categorical data (GH16793)

## 1.8 v0.20.2 (June 4, 2017)

This is a minor bug-fix release in the 0.20.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

### What's new in v0.20.2

- *Enhancements*
- *Performance Improvements*
- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Groupby/Resample/Rolling*
  - *Sparse*
  - *Reshaping*
  - *Numeric*
  - *Categorical*



### 1.8.1 Enhancements

- Unblocked access to additional compression types supported in pytables: ‘blosc:blosclz’, ‘blosc:lz4’, ‘blosc:lz4hc’, ‘blosc:snappy’, ‘blosc:zlib’, ‘blosc:zstd’ (GH14478)
- `Series` provides a `to_latex` method (GH16180)
- A new `groupby` method `ngroup()`, parallel to the existing `cumcount()`, has been added to return the group order (GH11642); see [here](#).

### 1.8.2 Performance Improvements

- Performance regression fix when indexing with a list-like (GH16285)
- Performance regression fix for `MultiIndex`s (GH16319, GH16346)
- Improved performance of `.clip()` with scalar arguments (GH15400)
- Improved performance of `groupby` with categorical groupers (GH16413)
- Improved performance of `MultiIndex.remove_unused_levels()` (GH16556)

### 1.8.3 Bug Fixes

- Silenced a warning on some Windows environments about “tput: terminal attributes: No such device or address” when detecting the terminal size. This fix only applies to python 3 (GH16496)
- Bug in using `pathlib.Path` or `py.path.local` objects with io functions (GH16291)
- Bug in `Index.symmetric_difference()` on two equal `MultiIndex`’s, results in a `TypeError` (GH13490)
- Bug in `DataFrame.update()` with `overwrite=False` and `NaN` values (GH15593)
- Passing an invalid engine to `read_csv()` now raises an informative `ValueError` rather than `UnboundLocalError`. (GH16511)
- Bug in `unique()` on an array of tuples (GH16519)
- Bug in `cut()` when labels are set, resulting in incorrect label ordering (GH16459)
- Fixed a compatibility issue with IPython 6.0’s tab completion showing deprecation warnings on `Categoricals` (GH16409)

#### 1.8.3.1 Conversion

- Bug in `to_numeric()` in which empty data inputs were causing a segfault of the interpreter (GH16302)
- Silence numpy warnings when broadcasting `DataFrame` to `Series` with comparison ops (GH16378, GH16306)

### 1.8.3.2 Indexing

- Bug in `DataFrame.reset_index(level=)` with single level index ([GH16263](#))
- Bug in partial string indexing with a monotonic, but not strictly-monotonic, index incorrectly reversing the slice bounds ([GH16515](#))
- Bug in `MultiIndex.remove_unused_levels()` that would not return a `MultiIndex` equal to the original. ([GH16556](#))

### 1.8.3.3 I/O

- Bug in `read_csv()` when `comment` is passed in a space delimited text file ([GH16472](#))
- Bug in `read_csv()` not raising an exception with nonexistent columns in `usecols` when it had the correct length ([GH14671](#))
- Bug that would force importing of the clipboard routines unnecessarily, potentially causing an import error on startup ([GH16288](#))
- Bug that raised `IndexError` when HTML-rendering an empty `DataFrame` ([GH15953](#))
- Bug in `read_csv()` in which `tarfile` object inputs were raising an error in Python 2.x for the C engine ([GH16530](#))
- Bug where `DataFrame.to_html()` ignored the `index_names` parameter ([GH16493](#))
- Bug where `pd.read_hdf()` returns numpy strings for index names ([GH13492](#))
- Bug in `HDFStore.select_as_multiple()` where `start/stop` arguments were not respected ([GH16209](#))

### 1.8.3.4 Plotting

- Bug in `DataFrame.plot` with a single column and a list-like `color` ([GH3486](#))
- Bug in `plot` where `NaT` in `DatetimeIndex` results in `Timestamp.min` ([GH12405](#))
- Bug in `DataFrame.boxplot` where `figsize` keyword was not respected for non-grouped boxplots ([GH11959](#))

### 1.8.3.5 Groupby/Resample/Rolling

- Bug in creating a time-based rolling window on an empty `DataFrame` ([GH15819](#))
- Bug in `rolling.cov()` with offset window ([GH16058](#))
- Bug in `.resample()` and `.groupby()` when aggregating on integers ([GH16361](#))

### 1.8.3.6 Sparse

- Bug in construction of `SparseDataFrame` from `scipy.sparse.dok_matrix` ([GH16179](#))

### 1.8.3.7 Reshaping

- Bug in `DataFrame.stack` with unsorted levels in `MultiIndex` columns ([GH16323](#))
- Bug in `pd.wide_to_long()` where no error was raised when `i` was not a unique identifier ([GH16382](#))
- Bug in `Series.isin(..)` with a list of tuples ([GH16394](#))
- Bug in construction of a `DataFrame` with mixed dtypes including an all-`NaT` column. ([GH16395](#))
- Bug in `DataFrame.agg()` and `Series.agg()` with aggregating on non-callable attributes ([GH16405](#))

### 1.8.3.8 Numeric

- Bug in `.interpolate()`, where `limit_direction` was not respected when `limit=None` (default) was passed ([GH16282](#))

### 1.8.3.9 Categorical

- Fixed comparison operations considering the order of the categories when both categoricals are unordered ([GH16014](#))

### 1.8.3.10 Other

- Bug in `DataFrame.drop()` with an empty-list with non-unique indices ([GH16270](#))

## 1.9 v0.20.1 (May 5, 2017)

This is a major release from 0.19.2 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- New `.agg()` API for `Series/DataFrame` similar to the `groupby-rolling-resample` API's, see [here](#)
- Integration with the `feather-format`, including a new top-level `pd.read_feather()` and `DataFrame.to_feather()` method, see [here](#).
- The `.ix` indexer has been deprecated, see [here](#)
- `Panel` has been deprecated, see [here](#)
- Addition of an `IntervalIndex` and `Interval` scalar type, see [here](#)
- Improved user API when grouping by index levels in `.groupby()`, see [here](#)
- Improved support for `UInt64` dtypes, see [here](#)
- A new orient for JSON serialization, `orient='table'`, that uses the Table Schema spec and that gives the possibility for a more interactive repr in the Jupyter Notebook, see [here](#)
- Experimental support for exporting styled `DataFrames` (`DataFrame.style`) to Excel, see [here](#)
- Window binary `corr/cov` operations now return a `MultiIndexed DataFrame` rather than a `Panel`, as `Panel` is now deprecated, see [here](#)
- Support for S3 handling now uses `s3fs`, see [here](#)

- Google BigQuery support now uses the `pandas-gbq` library, see [here](#)

**Warning:** Pandas has changed the internal structure and layout of the codebase. This can affect imports that are not from the top-level `pandas.*` namespace, please see the changes [here](#).

Check the *API Changes* and *deprecations* before updating.

**Note:** This is a combined release for 0.20.0 and 0.20.1. Version 0.20.1 contains one additional change for backwards-compatibility with downstream projects using pandas' `utils` routines. ([GH16250](#))

---

### What's new in v0.20.0

- *New features*
  - *agg API for DataFrame/Series*
  - *dtype keyword for data IO*
  - *.to\_datetime() has gained an origin parameter*
  - *Groupby Enhancements*
  - *Better support for compressed URLs in read\_csv*
  - *Pickle file I/O now supports compression*
  - *UInt64 Support Improved*
  - *GroupBy on Categoricals*
  - *Table Schema Output*
  - *SciPy sparse matrix from/to SparseDataFrame*
  - *Excel output for styled DataFrames*
  - *IntervalIndex*
  - *Other Enhancements*
- *Backwards incompatible API changes*
  - *Possible incompatibility for HDF5 formats created with pandas < 0.13.0*
  - *Map on Index types now return other Index types*
  - *Accessing datetime fields of Index now return Index*
  - *pd.unique will now be consistent with extension types*
  - *S3 File Handling*
  - *Partial String Indexing Changes*
  - *Concat of different float dtypes will not automatically upcast*
  - *Pandas Google BigQuery support has moved*
  - *Memory Usage for Index is more Accurate*
  - *DataFrame.sort\_index changes*

- *Groupby Describe Formatting*
- *Window Binary Corr/Cov operations return a MultiIndex DataFrame*
- *HDFStore where string comparison*
- *Index.intersection and inner join now preserve the order of the left Index*
- *Pivot Table always returns a DataFrame*
- *Other API Changes*
- *Reorganization of the library: Privacy Changes*
  - *Modules Privacy Has Changed*
  - `pandas.errors`
  - `pandas.testing`
  - `pandas.plotting`
  - *Other Development Changes*
- *Deprecations*
  - *Deprecate .ix*
  - *Deprecate Panel*
  - *Deprecate groupby.agg() with a dictionary when renaming*
  - *Deprecate .plotting*
  - *Other Deprecations*
- *Removal of prior version deprecations/changes*
- *Performance Improvements*
- *Bug Fixes*
  - *Conversion*
  - *Indexing*
  - *I/O*
  - *Plotting*
  - *Groupby/Resample/Rolling*
  - *Sparse*
  - *Reshaping*
  - *Numeric*
  - *Other*

## 1.9.1 New features

### 1.9.1.1 agg API for DataFrame/Series

Series & DataFrame have been enhanced to support the aggregation API. This is a familiar API from groupby, window operations, and resampling. This allows aggregation operations in a concise way by using `agg()` and

`transform()`. The full documentation is [here](#) (GH1623).

Here is a sample

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...:                      index=pd.date_range('1/1/2000', periods=10))
...:

In [2]: df.iloc[3:7] = np.nan

In [3]: df
Out[3]:
```

	A	B	C
2000-01-01	1.682600	0.413582	1.689516
2000-01-02	-2.099110	-1.180182	1.595661
2000-01-03	-0.419048	0.522165	-1.208946
2000-01-04	NaN	NaN	NaN
2000-01-05	NaN	NaN	NaN
2000-01-06	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN
2000-01-08	0.955435	-0.133009	2.011466
2000-01-09	0.578780	0.897126	-0.980013
2000-01-10	-0.045748	0.361601	-0.208039

One can operate using string function names, callables, lists, or dictionaries of these.

Using a single function is equivalent to `.apply`.

```
In [4]: df.agg('sum')
Out[4]:
```

A	0.652908
B	0.881282
C	2.899645

dtype: float64

Multiple aggregations with a list of functions.

```
In [5]: df.agg(['sum', 'min'])
Out[5]:
```

	A	B	C
sum	0.652908	0.881282	2.899645
min	-2.099110	-1.180182	-1.208946

Using a dict provides the ability to apply specific aggregations per column. You will get a matrix-like output of all of the aggregators. The output has one column per unique function. Those functions applied to a particular column will be NaN:

```
In [6]: df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
Out[6]:
```

	A	B
max	NaN	0.897126
min	-2.099110	-1.180182
sum	0.652908	NaN

The API also supports a `.transform()` function for broadcasting results.

```
In [7]: df.transform(['abs', lambda x: x - x.min()])
Out[7]:
```

(continues on next page)

(continued from previous page)

	A		B		C	
	abs	<lambda>	abs	<lambda>	abs	<lambda>
2000-01-01	1.682600	3.781710	0.413582	1.593764	1.689516	2.898461
2000-01-02	2.099110	0.000000	1.180182	0.000000	1.595661	2.804606
2000-01-03	0.419048	1.680062	0.522165	1.702346	1.208946	0.000000
2000-01-04	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-05	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-06	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN	NaN	NaN
2000-01-08	0.955435	3.054545	0.133009	1.047173	2.011466	3.220412
2000-01-09	0.578780	2.677890	0.897126	2.077307	0.980013	0.228932
2000-01-10	0.045748	2.053362	0.361601	1.541782	0.208039	1.000907

When presented with mixed dtypes that cannot be aggregated, `.agg()` will only take the valid aggregations. This is similar to how `groupby .agg()` works. (GH15015)

```
In [8]: df = pd.DataFrame({'A': [1, 2, 3],
...:                      'B': [1., 2., 3.],
...:                      'C': ['foo', 'bar', 'baz'],
...:                      'D': pd.date_range('20130101', periods=3)})
...:

In [9]: df.dtypes
Out[9]:
A          int64
B          float64
C          object
D    datetime64[ns]
dtype: object
```

```
In [10]: df.agg(['min', 'sum'])
Out[10]:
   A    B      C      D
min 1  1.0    bar 2013-01-01
sum 6  6.0 foobarbaz      NaT
```

### 1.9.1.2 dtype keyword for data IO

The 'python' engine for `read_csv()`, as well as the `read_fwf()` function for parsing fixed-width text files and `read_excel()` for parsing Excel files, now accept the `dtype` keyword argument for specifying the types of specific columns (GH14295). See the *io docs* for more information.

```
In [11]: data = "a  b\n1  2\n3  4"

In [12]: pd.read_fwf(StringIO(data)).dtypes
Out[12]:
a    int64
b    int64
dtype: object

In [13]: pd.read_fwf(StringIO(data), dtype={'a': 'float64', 'b': 'object'}).dtypes
Out[13]:
a    float64
b    object
dtype: object
```

### 1.9.1.3 .to\_datetime() has gained an origin parameter

`to_datetime()` has gained a new parameter, `origin`, to define a reference date from where to compute the resulting timestamps when parsing numerical values with a specific `unit` specified. (GH11276, GH11745)

For example, with 1960-01-01 as the starting date:

```
In [14]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[14]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype=
↳ 'datetime64[ns]', freq=None)
```

The default is set at `origin='unix'`, which defaults to 1970-01-01 00:00:00, which is commonly called ‘unix epoch’ or POSIX time. This was the previous default, so this is a backward compatible change.

```
In [15]: pd.to_datetime([1, 2, 3], unit='D')
Out[15]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype=
↳ 'datetime64[ns]', freq=None)
```

### 1.9.1.4 Groupby Enhancements

Strings passed to `DataFrame.groupby()` as the `by` parameter may now reference either column names or index level names. Previously, only column names could be referenced. This allows to easily group by a column and index level at the same time. (GH5677)

```
In [16]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
...:               ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
...:
```

```
In [17]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])
```

```
In [18]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
.....:                      'B': np.arange(8)},
.....:                      index=index)
.....:
```

```
In [19]: df
```

```
Out [19]:
```

		A	B
first	second		
bar	one	1	0
	two	1	1
baz	one	1	2
	two	1	3
foo	one	2	4
	two	2	5
qux	one	3	6
	two	3	7

```
In [20]: df.groupby(['second', 'A']).sum()
```

		B
second	A	
one	1	2
	2	4
	3	6

(continues on next page)



(continued from previous page)

```

two    1  4
       2  5
       3  7

```

### 1.9.1.5 Better support for compressed URLs in `read_csv`

The compression code was refactored (GH12688). As a result, reading dataframes from URLs in `read_csv()` or `read_table()` now supports additional compression methods: `xz`, `bz2`, and `zip` (GH14570). Previously, only `gzip` compression was supported. By default, compression of URLs and paths are now inferred using their file extensions. Additionally, support for `bz2` compression in the python 2 C-engine improved (GH14874).

```

In [21]: url = 'https://github.com/{repo}/raw/{branch}/{path}'.format(
.....:     repo = 'pandas-dev/pandas',
.....:     branch = 'master',
.....:     path = 'pandas/tests/io/parser/data/salaries.csv.bz2',
.....: )
.....:

In [22]: df = pd.read_table(url, compression='infer') # default, infer compression

In [23]: df = pd.read_table(url, compression='bz2') # explicitly specify compression

In [24]: df.head(2)
Out[24]:
      S  X  E  M
0  13876  1  1  1
1  11608  1  3  0

```

### 1.9.1.6 Pickle file I/O now supports compression

`read_pickle()`, `DataFrame.to_pickle()` and `Series.to_pickle()` can now read from and write to compressed pickle files. Compression methods can be an explicit parameter or be inferred from the file extension. See [the docs here](#).

```

In [25]: df = pd.DataFrame({
.....:     'A': np.random.randn(1000),
.....:     'B': 'foo',
.....:     'C': pd.date_range('20130101', periods=1000, freq='s')})
.....:

```

Using an explicit compression type

```

In [26]: df.to_pickle("data.pkl.compress", compression="gzip")

In [27]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [28]: rt.head()
Out[28]:
      A      B      C
0  1.578227  foo  2013-01-01 00:00:00
1 -0.230575  foo  2013-01-01 00:00:01
2  0.695530  foo  2013-01-01 00:00:02
3 -0.466001  foo  2013-01-01 00:00:03
4 -0.154972  foo  2013-01-01 00:00:04

```

The default is to infer the compression type from the extension (`compression='infer'`):

```
In [29]: df.to_pickle("data.pkl.gz")

In [30]: rt = pd.read_pickle("data.pkl.gz")

In [31]: rt.head()
Out[31]:
```

	A	B	C
0	1.578227	foo	2013-01-01 00:00:00
1	-0.230575	foo	2013-01-01 00:00:01
2	0.695530	foo	2013-01-01 00:00:02
3	-0.466001	foo	2013-01-01 00:00:03
4	-0.154972	foo	2013-01-01 00:00:04

```
In [32]: df["A"].to_pickle("s1.pkl.bz2")

In [33]: rt = pd.read_pickle("s1.pkl.bz2")

In [34]: rt.head()
Out[34]:
```

0	1.578227
1	-0.230575
2	0.695530
3	-0.466001
4	-0.154972

```
Name: A, dtype: float64
```

### 1.9.1.7 UInt64 Support Improved

Pandas has significantly improved support for operations involving unsigned, or purely non-negative, integers. Previously, handling these integers would result in improper rounding or data-type casting, leading to incorrect results. Notably, a new numerical index, `UInt64Index`, has been created ([GH14937](#))

```
In [35]: idx = pd.UInt64Index([1, 2, 3])

In [36]: df = pd.DataFrame({'A': ['a', 'b', 'c']}, index=idx)

In [37]: df.index
Out[37]: UInt64Index([1, 2, 3], dtype='uint64')
```

- Bug in converting object elements of array-like objects to unsigned 64-bit integers ([GH4471](#), [GH14982](#))
- Bug in `Series.unique()` in which unsigned 64-bit integers were causing overflow ([GH14721](#))
- Bug in `DataFrame` construction in which unsigned 64-bit integer elements were being converted to objects ([GH14881](#))
- Bug in `pd.read_csv()` in which unsigned 64-bit integer elements were being improperly converted to the wrong data types ([GH14983](#))
- Bug in `pd.unique()` in which unsigned 64-bit integers were causing overflow ([GH14915](#))
- Bug in `pd.value_counts()` in which unsigned 64-bit integers were being erroneously truncated in the output ([GH14934](#))

### 1.9.1.8 GroupBy on Categoricals

In previous versions, `.groupby(..., sort=False)` would fail with a `ValueError` when grouping on a categorical series with some categories not appearing in the data. (GH13179)

```
In [38]: chromosomes = np.r_[np.arange(1, 23).astype(str), ['X', 'Y']]

In [39]: df = pd.DataFrame({
.....:     'A': np.random.randint(100),
.....:     'B': np.random.randint(100),
.....:     'C': np.random.randint(100),
.....:     'chromosomes': pd.Categorical(np.random.choice(chromosomes, 100),
.....:                                     categories=chromosomes,
.....:                                     ordered=True)})

In [40]: df
Out[40]:
```

	A	B	C	chromosomes
0	80	36	94	12
1	80	36	94	X
2	80	36	94	19
3	80	36	94	22
4	80	36	94	17
5	80	36	94	6
6	80	36	94	13
..	..	..	..	...
93	80	36	94	21
94	80	36	94	20
95	80	36	94	11
96	80	36	94	16
97	80	36	94	21
98	80	36	94	18
99	80	36	94	8

[100 rows x 4 columns]

#### Previous Behavior:

```
In [3]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
-----
ValueError: items in new_categories are not the same as in old categories
```

#### New Behavior:

```
In [41]: df[df.chromosomes != '1'].groupby('chromosomes', sort=False).sum()
Out[41]:
```

	A	B	C
chromosomes			
2	320	144	376
3	400	180	470
4	240	108	282
5	240	108	282
6	400	180	470
7	400	180	470
8	480	216	564
...	...	...	...
19	400	180	470

(continues on next page)

(continued from previous page)

```

20          160    72   188
21          480   216   564
22          160    72   188
X           400   180   470
Y           320   144   376
1              0     0     0

```

```
[24 rows x 3 columns]
```

### 1.9.1.9 Table Schema Output

The new orient `'table'` for `DataFrame.to_json()` will generate a [Table Schema](#) compatible string representation of the data.

```

In [42]: df = pd.DataFrame(
.....:     {'A': [1, 2, 3],
.....:      'B': ['a', 'b', 'c'],
.....:      'C': pd.date_range('2016-01-01', freq='d', periods=3),
.....:     }, index=pd.Index(range(3), name='idx'))
.....:

```

```
In [43]: df
```

```

Out[43]:
   A  B      C
idx
0   1  a 2016-01-01
1   2  b 2016-01-02
2   3  c 2016-01-03

```

```
In [44]: df.to_json(orient='table')
```

```

////////////////////////////////////
↪ '{"schema": {"fields": [{"name": "idx", "type": "integer"}, {"name": "A", "type": "integer"},
↪ {"name": "B", "type": "string"}, {"name": "C", "type": "datetime"}], "primaryKey": ["idx"],
↪ "pandas_version": "0.20.0"}, "data": [{"idx": 0, "A": 1, "B": "a", "C": "2016-01-
↪ 01T00:00:00.000Z"}, {"idx": 1, "A": 2, "B": "b", "C": "2016-01-02T00:00:00.000Z"}, {"idx": 2,
↪ "A": 3, "B": "c", "C": "2016-01-03T00:00:00.000Z"}]}'

```

See [IO: Table Schema for more information](#).

Additionally, the repr for `DataFrame` and `Series` can now publish this JSON Table schema representation of the Series or DataFrame if you are using IPython (or another frontend like [interact](#) using the Jupyter messaging protocol). This gives frontends like the Jupyter notebook and [interact](#) more flexibility in how they display pandas objects, since they have more information about the data. You must enable this by setting the `display.html.table_schema` option to `True`.

### 1.9.1.10 SciPy sparse matrix from/to SparseDataFrame

Pandas now supports creating sparse dataframes directly from `scipy.sparse.spmatrix` instances. See the [documentation](#) for more information. ([GH4343](#))

All sparse formats are supported, but matrices that are not in `COOrdinate` format will be converted, copying data as needed.

```

In [45]: from scipy.sparse import csr_matrix

In [46]: arr = np.random.random(size=(1000, 5))

In [47]: arr[arr < .9] = 0

In [48]: sp_arr = csr_matrix(arr)

In [49]: sp_arr
Out[49]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
      with 521 stored elements in Compressed Sparse Row format>

In [50]: sdf = pd.SparseDataFrame(sp_arr)

In [51]: sdf
Out[51]:
   0      1      2      3      4
0  NaN    NaN    NaN    NaN    NaN
1  NaN    NaN    NaN  0.955103  NaN
2  NaN    NaN    NaN  0.900469  NaN
3  NaN    NaN    NaN    NaN    NaN
4  NaN  0.924771    NaN    NaN    NaN
5  NaN    NaN    NaN    NaN    NaN
6  NaN    NaN    NaN    NaN    NaN
..  ..      ...      ...      ...      ...
993 NaN    NaN    NaN    NaN    NaN
994 NaN    NaN    NaN    NaN  0.972191
995 NaN  0.979898  0.97901    NaN    NaN
996 NaN    NaN    NaN    NaN    NaN
997 NaN    NaN    NaN    NaN    NaN
998 NaN    NaN    NaN    NaN    NaN
999 NaN    NaN    NaN    NaN    NaN

[1000 rows x 5 columns]

```

To convert a `SparseDataFrame` back to sparse SciPy matrix in COO format, you can use:

```

In [52]: sdf.to_coo()
Out[52]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
      with 521 stored elements in COOrdinate format>

```

### 1.9.1.11 Excel output for styled DataFrames

Experimental support has been added to export `DataFrame.style` formats to Excel using the `openpyxl` engine. (GH15530)

For example, after running the following, `styled.xlsx` renders as below:

```

In [53]: np.random.seed(24)

In [54]: df = pd.DataFrame({'A': np.linspace(1, 10, 10)})

In [55]: df = pd.concat([df, pd.DataFrame(np.random.RandomState(24).randn(10, 4),
      ....:                                     columns=list('BCDE'))],

```

(continues on next page)

(continued from previous page)

```

.....:                 axis=1)
.....:

In [56]: df.iloc[0, 2] = np.nan

In [57]: df
Out [57]:
   A         B         C         D         E
0  1.0  1.329212      NaN -0.316280 -0.990810
1  2.0 -1.070816 -1.438713  0.564417  0.295722
2  3.0 -1.626404  0.219565  0.678805  1.889273
3  4.0  0.961538  0.104011 -0.481165  0.850229
4  5.0  1.453425  1.057737  0.165562  0.515018
5  6.0 -1.336936  0.562861  1.392855 -0.063328
6  7.0  0.121668  1.207603 -0.002040  1.627796
7  8.0  0.354493  1.037528 -0.385684  0.519818
8  9.0  1.686583 -1.325963  1.428984 -2.089354
9 10.0 -0.129820  0.631523 -0.586538  0.290720

In [58]: styled = df.style.\
.....:     applymap(lambda val: 'color: %s' % 'red' if val < 0 else 'black').\
.....:     highlight_max()
.....:

In [59]: styled.to_excel('styled.xlsx', engine='openpyxl')

```

	A	B	C	D	E	F
1		A	B	C	D	E
2	0	1	1.329212		-0.31628	-0.99081
3	1	2	-1.070816	-1.438713	0.564417	0.295722
4	2	3	-1.626404	0.219565	0.678805	1.889273
5	3	4	0.961538	0.104011	-0.481165	0.850229
6	4	5	1.453425	1.057737	0.165562	0.515018
7	5	6	-1.336936	0.562861	1.392855	-0.063328
8	6	7	0.121668	1.207603	-0.00204	1.627796
9	7	8	0.354493	1.037528	-0.385684	0.519818
10	8	9	1.686583	-1.325963	1.428984	-2.089354
11	9	10	-0.12982	0.631523	-0.586538	0.29072

See the [Style documentation](#) for more detail.

### 1.9.1.12 IntervalIndex

pandas has gained an `IntervalIndex` with its own dtype, `interval` as well as the `Interval` scalar type. These allow first-class support for interval notation, specifically as a return type for the categories in `cut()` and `qcut()`. The `IntervalIndex` allows some unique indexing, see the [docs](#). ([GH7640](#), [GH8625](#))

The returned categories were strings, representing Intervals

New behavior:

Furthermore, this allows one to bin *other* data with these same bins, with `NaN` representing a missing value similar to other dtypes.

An IntervalIndex can also be used in Series and DataFrame as the index.

### 1.9. v0.20.1 (May 5, 2017)

Selecting via a specific interval:

```
In [66]: df.loc[pd.Interval(1.5, 3.0)]
Out [66]:
A      1
Name: (1.5, 3.0], dtype: int64
```

Selecting via a scalar value that is contained *in* the intervals.

```
In [67]: df.loc[0]
Out [67]:
      A
B
(-0.003, 1.5]  0
(-0.003, 1.5]  2
(-0.003, 1.5]  3
```

### 1.9.1.13 Other Enhancements

- `DataFrame.rolling()` now accepts the parameter `closed='right'|'left'|'both'|'neither'` to choose the rolling window-endpoint closedness. See the [documentation](#) (GH13965)
- Integration with the feather-format, including a new top-level `pd.read_feather()` and `DataFrame.to_feather()` method, see [here](#).
- `Series.str.replace()` now accepts a callable, as replacement, which is passed to `re.sub` (GH15055)
- `Series.str.replace()` now accepts a compiled regular expression as a pattern (GH15446)
- `Series.sort_index` accepts parameters `kind` and `na_position` (GH13589, GH14444)
- `DataFrame` and `DataFrame.groupby()` have gained a `nunique()` method to count the distinct values over an axis (GH14336, GH15197).
- `DataFrame` has gained a `melt()` method, equivalent to `pd.melt()`, for unpivoting from a wide to long format (GH12640).
- `pd.read_excel()` now preserves sheet order when using `sheetname=None` (GH9930)
- Multiple offset aliases with decimal points are now supported (e.g. `0.5min` is parsed as `30s`) (GH8419)
- `.isnull()` and `.notnull()` have been added to `Index` object to make them more consistent with the `Series` API (GH15300)
- New `UnsortedIndexError` (subclass of `KeyError`) raised when indexing/slicing into an unsorted `MultiIndex` (GH11897). This allows differentiation between errors due to lack of sorting or an incorrect key. See [here](#)
- `MultiIndex` has gained a `.to_frame()` method to convert to a `DataFrame` (GH12397)
- `pd.cut` and `pd.qcut` now support `datetime64` and `timedelta64` dtypes (GH14714, GH14798)
- `pd.qcut` has gained the `duplicates='raise'|'drop'` option to control whether to raise on duplicated edges (GH7751)
- `Series` provides a `to_excel` method to output Excel files (GH8825)
- The `usecols` argument in `pd.read_csv()` now accepts a callable function as a value (GH14154)
- The `skiprows` argument in `pd.read_csv()` now accepts a callable function as a value (GH10882)
- The `nrows` and `chunksize` arguments in `pd.read_csv()` are supported if both are passed (GH6774, GH15755)



- `DataFrame.plot` now prints a title above each subplot if `suplots=True` and `title` is a list of strings ([GH14753](#))
- `DataFrame.plot` can pass the matplotlib 2.0 default color cycle as a single string as color parameter, see [here](#). ([GH15516](#))
- `Series.interpolate()` now supports `timedelta` as an index type with `method='time'` ([GH6424](#))
- Addition of a `level` keyword to `DataFrame/Series.rename` to rename labels in the specified level of a `MultiIndex` ([GH4160](#)).
- `DataFrame.reset_index()` will now interpret a tuple `index.name` as a key spanning across levels of columns, if this is a `MultiIndex` ([GH16164](#))
- `Timedelta.isoformat` method added for formatting `Timedeltas` as an [ISO 8601 duration](#). See the [Timedelta docs](#) ([GH15136](#))
- `.select_dtypes()` now allows the string `datetimez` to generically select datetimes with `tz` ([GH14910](#))
- The `.to_latex()` method will now accept `multicolumn` and `multirow` arguments to use the accompanying LaTeX enhancements
- `pd.merge_asof()` gained the option `direction='backward'|'forward'|'nearest'` ([GH14887](#))
- `Series/DataFrame.asfreq()` have gained a `fill_value` parameter, to fill missing values ([GH3715](#)).
- `Series/DataFrame.resample.asfreq` have gained a `fill_value` parameter, to fill missing values during resampling ([GH3715](#)).
- `pandas.util.hash_pandas_object()` has gained the ability to hash a `MultiIndex` ([GH15224](#))
- `Series/DataFrame.squeeze()` have gained the `axis` parameter. ([GH15339](#))
- `DataFrame.to_excel()` has a new `freeze_panes` parameter to turn on Freeze Panes when exporting to Excel ([GH15160](#))
- `pd.read_html()` will parse multiple header rows, creating a `MultiIndex` header. ([GH13434](#)).
- HTML table output skips `colspan` or `rowspan` attribute if equal to 1. ([GH15403](#))
- `pandas.io.formats.style.Styler` template now has blocks for easier extension, see the [example notebook](#) ([GH15649](#))
- `Styler.render()` now accepts `**kwargs` to allow user-defined variables in the template ([GH15649](#))
- Compatibility with Jupyter notebook 5.0; `MultiIndex` column labels are left-aligned and `MultiIndex` row-labels are top-aligned ([GH15379](#))
- `TimedeltaIndex` now has a custom date-tick formatter specifically designed for nanosecond level precision ([GH8711](#))
- `pd.api.types.union_categoricals` gained the `ignore_ordered` argument to allow ignoring the ordered attribute of unioned categoricals ([GH13410](#)). See the [categorical union docs](#) for more information.
- `DataFrame.to_latex()` and `DataFrame.to_string()` now allow optional header aliases. ([GH15536](#))
- Re-enable the `parse_dates` keyword of `pd.read_excel()` to parse string columns as dates ([GH14326](#))
- Added `.empty` property to subclasses of `Index`. ([GH15270](#))
- Enabled floor division for `Timedelta` and `TimedeltaIndex` ([GH15828](#))
- `pandas.io.json.json_normalize()` gained the option `errors='ignore'|'raise'`; the default is `errors='raise'` which is backward compatible. ([GH14583](#))

- `pandas.io.json.json_normalize()` with an empty list will return an empty DataFrame (GH15534)
- `pandas.io.json.json_normalize()` has gained a `sep` option that accepts `str` to separate joined fields; the default is “.”, which is backward compatible. (GH14883)
- `MultiIndex.remove_unused_levels()` has been added to facilitate *removing unused levels*. (GH15694)
- `pd.read_csv()` will now raise a `ParserError` error whenever any parsing error occurs (GH15913, GH15925)
- `pd.read_csv()` now supports the `error_bad_lines` and `warn_bad_lines` arguments for the Python parser (GH15925)
- The `display.show_dimensions` option can now also be used to specify whether the length of a `Series` should be shown in its repr (GH7117).
- `parallel_coordinates()` has gained a `sort_labels` keyword argument that sorts class labels and the colors assigned to them (GH15908)
- Options added to allow one to turn on/off using `bottleneck` and `numexpr`, see [here](#) (GH16157)
- `DataFrame.style.bar()` now accepts two more options to further customize the bar chart. Bar alignment is set with `align='left'|'mid'|'zero'`, the default is “left”, which is backward compatible; You can now pass a list of `color=[color_negative, color_positive]`. (GH14757)

## 1.9.2 Backwards incompatible API changes

### 1.9.2.1 Possible incompatibility for HDF5 formats created with pandas < 0.13.0

`pd.TimeSeries` was deprecated officially in 0.17.0, though has already been an alias since 0.13.0. It has been dropped in favor of `pd.Series`. (GH15098).

This *may* cause HDF5 files that were created in prior versions to become unreadable if `pd.TimeSeries` was used. This is most likely to be for pandas < 0.13.0. If you find yourself in this situation. You can use a recent prior version of pandas to read in your HDF5 files, then write them out again after applying the procedure below.

```
In [2]: s = pd.TimeSeries([1,2,3], index=pd.date_range('20130101', periods=3))

In [3]: s
Out[3]:
2013-01-01    1
2013-01-02    2
2013-01-03    3
Freq: D, dtype: int64

In [4]: type(s)
Out[4]: pandas.core.series.TimeSeries

In [5]: s = pd.Series(s)

In [6]: s
Out[6]:
2013-01-01    1
2013-01-02    2
2013-01-03    3
Freq: D, dtype: int64
```

(continues on next page)



```
In [76]: s = Series(date_range('2011-01-02T00:00', '2011-01-02T02:00', freq='H').tz_
↳ localize('Asia/Tokyo'))

In [77]: s
Out[77]:
0    2011-01-02 00:00:00+09:00
1    2011-01-02 01:00:00+09:00
2    2011-01-02 02:00:00+09:00
dtype: datetime64[ns, Asia/Tokyo]
```

Previous Behavior:

```
In [9]: s.map(lambda x: x.hour)
Out[9]:
0    0
1    1
2    2
dtype: int32
```

New Behavior:

```
In [78]: s.map(lambda x: x.hour)
Out[78]:
0    0
1    1
2    2
dtype: int64
```

### 1.9.2.3 Accessing datetime fields of Index now return Index

The datetime-related attributes (see [here](#) for an overview) of `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex` previously returned numpy arrays. They will now return a new `Index` object, except in the case of a boolean field, where the result will still be a boolean ndarray. (GH15022)

Previous behaviour:

```
In [1]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [2]: idx.hour
Out[2]: array([ 0, 10, 20,  6, 16], dtype=int32)
```

New Behavior:

```
In [79]: idx = pd.date_range("2015-01-01", periods=5, freq='10H')

In [80]: idx.hour
Out[80]: Int64Index([0, 10, 20, 6, 16], dtype='int64')
```

This has the advantage that specific `Index` methods are still available on the result. On the other hand, this might have backward incompatibilities: e.g. compared to numpy arrays, `Index` objects are not mutable. To get the original ndarray, you can always convert explicitly using `np.asarray(idx.hour)`.

### 1.9.2.4 `pd.unique` will now be consistent with extension types

In prior versions, using `Series.unique()` and `pandas.unique()` on Categorical and tz-aware data-types would yield different return types. These are now made consistent. (GH15903)

- Datetime tz-aware

Previous behaviour:

```
# Series
In [5]: pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                  pd.Timestamp('20160101', tz='US/Eastern')]).unique()
Out [5]: array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
               ↪dtype=object)

In [6]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                             pd.Timestamp('20160101', tz='US/Eastern')]))
Out [6]: array(['2016-01-01T05:00:00.000000000'], dtype='datetime64[ns]')

# Index
In [7]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
                  pd.Timestamp('20160101', tz='US/Eastern')]).unique()
Out [7]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/
               ↪Eastern]', freq=None)

In [8]: pd.unique([pd.Timestamp('20160101', tz='US/Eastern'),
                  pd.Timestamp('20160101', tz='US/Eastern')])
Out [8]: array(['2016-01-01T05:00:00.000000000'], dtype='datetime64[ns]')
```

New Behavior:

```
# Series, returns an array of Timestamp tz-aware
In [81]: pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                  ....:      pd.Timestamp('20160101', tz='US/Eastern')]).unique()
                  ....:
Out [81]: array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
               ↪dtype=object)

In [82]: pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
                  ....:      pd.Timestamp('20160101', tz='US/Eastern')]))
                  ....:
Out [82]: array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')], dtype=object)

# Index, returns a DatetimeIndex
In [83]: pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
                  ....:      pd.Timestamp('20160101', tz='US/Eastern')]).unique()
                  ....:
Out [83]: DatetimeIndex(['2016-01-01 05:00:00-05:00'], dtype='datetime64[ns, US/Eastern]',
               ↪freq=None)

In [84]: pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
                  ....:      pd.Timestamp('20160101', tz='US/Eastern')]))
                  ....:
Out [84]: DatetimeIndex(['2016-01-01 00:00:00-05:00'], dtype='datetime64[ns, US/Eastern]',
               ↪freq=None)
```

- Categoricals

Previous behaviour:



### 1.9.2.7 Concat of different float dtypes will not automatically upcast

Previously, concat of multiple objects with different float dtypes would automatically upcast results to a dtype of float64. Now the smallest acceptable dtype will be used ([GH13247](#))

```
In [88]: df1 = pd.DataFrame(np.array([1.0], dtype=np.float32, ndmin=2))

In [89]: df1.dtypes
Out[89]:
0    float32
dtype: object

In [90]: df2 = pd.DataFrame(np.array([np.nan], dtype=np.float32, ndmin=2))

In [91]: df2.dtypes
Out[91]:
0    float32
dtype: object
```

Previous Behavior:

```
In [7]: pd.concat([df1, df2]).dtypes
Out[7]:
0    float64
dtype: object
```

New Behavior:

```
In [92]: pd.concat([df1, df2]).dtypes
Out[92]:
0    float32
dtype: object
```

### 1.9.2.8 Pandas Google BigQuery support has moved

pandas has split off Google BigQuery support into a separate package `pandas-gbq`. You can `conda install pandas-gbq -c conda-forge` or `pip install pandas-gbq` to get it. The functionality of `read_gbq()` and `DataFrame.to_gbq()` remain the same with the currently released version of `pandas-gbq=0.1.4`. Documentation is now hosted [here](#) ([GH15347](#))

### 1.9.2.9 Memory Usage for Index is more Accurate

In previous versions, showing `.memory_usage()` on a pandas structure that has an index, would only include actual index values and not include structures that facilitated fast indexing. This will generally be different for `Index` and `MultiIndex` and less-so for other index types. ([GH15237](#))

Previous Behavior:

```
In [8]: index = Index(['foo', 'bar', 'baz'])

In [9]: index.memory_usage(deep=True)
Out[9]: 180

In [10]: index.get_loc('foo')
Out[10]: 0
```

(continues on next page)

(continued from previous page)

```
In [11]: index.memory_usage(deep=True)
Out[11]: 180
```

New Behavior:

```
In [8]: index = Index(['foo', 'bar', 'baz'])

In [9]: index.memory_usage(deep=True)
Out[9]: 180

In [10]: index.get_loc('foo')
Out[10]: 0

In [11]: index.memory_usage(deep=True)
Out[11]: 260
```

### 1.9.2.10 DataFrame.sort\_index changes

In certain cases, calling `.sort_index()` on a MultiIndexed DataFrame would return the *same* DataFrame without seeming to sort. This would happen with a lexicographically sorted, but non-monotonic levels. ([GH15622](#), [GH15687](#), [GH14015](#), [GH13431](#), [GH15797](#))

This is *unchanged* from prior versions, but shown for illustration purposes:

```
In [93]: df = DataFrame(np.arange(6), columns=['value'], index=MultiIndex.from_
↳product([list('BA'), range(3)]))

In [94]: df
Out[94]:
   value
B 0      0
  1      1
  2      2
A 0      3
  1      4
  2      5
```

```
In [95]: df.index.is_lexsorted()
Out[95]: False

In [96]: df.index.is_monotonic
Out[96]: False
```

Sorting works as expected

```
In [97]: df.sort_index()
Out[97]:
   value
A 0      3
  1      4
  2      5
B 0      0
  1      1
  2      2
```



```
In [98]: df.sort_index().index.is_lexsorted()
Out[98]: True
```

```
In [99]: df.sort_index().index.is_monotonic
Out[99]: True
```

However, this example, which has a non-monotonic 2nd level, doesn't behave as desired.

```
In [100]: df = pd.DataFrame(
.....:     {'value': [1, 2, 3, 4]},
.....:     index=pd.MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
.....:                           labels=[[0, 0, 1, 1], [0, 1, 0, 1]]))
.....:
```

```
In [101]: df
```

```
Out[101]:
      value
a bb      1
  aa      2
b bb      3
  aa      4
```

Previous Behavior:

```
In [11]: df.sort_index()
Out[11]:
      value
a bb      1
  aa      2
b bb      3
  aa      4

In [14]: df.sort_index().index.is_lexsorted()
Out[14]: True

In [15]: df.sort_index().index.is_monotonic
Out[15]: False
```

New Behavior:

```
In [102]: df.sort_index()
Out[102]:
      value
a aa      2
  bb      1
b aa      4
  bb      3

In [103]: df.sort_index().index.is_lexsorted()
Out[103]: True

In [104]: df.sort_index().index.is_monotonic
Out[104]: False
```

### 1.9.2.11 Groupby Describe Formatting

The output formatting of `groupby.describe()` now labels the `describe()` metrics in the columns instead of the index. This format is consistent with `groupby.agg()` when applying multiple functions at once. (GH4792)

Previous Behavior:

```
In [1]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})
```

```
In [2]: df.groupby('A').describe()
```

Out [2] :

B

A

```
1 count 2.000000
```

```
mean    1.500000
```

std	0.707107
-----	----------

min	1.000000
-----	----------

25%	1.250000
-----	----------

50%	1.500000
-----	----------

75%	1.750000
-----	----------

max	2.000000
-----	----------

```
2 count 2.000000
```

```
mean    3.500000
```

std	0.707107
-----	----------

min	3.000000
-----	----------

25%	3.250000
-----	----------

50%	3.500000
-----	----------

75%	3.750000
-----	----------

max	4.000000
-----	----------

```
In [3]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
```

Out [3] :

	B			
mean		std	amin	amax

A

1	1.5	0.707107	1	2
---	-----	----------	---	---

2	3.5	0.707107	3	4
---	-----	----------	---	---

**New Behavior:**

```
In [105]: df = pd.DataFrame({'A': [1, 1, 2, 2], 'B': [1, 2, 3, 4]})
```

```
In [106]: df.groupby('A').describe()
```

Out [106] :

B							
count	mean	std	min	25%	50%	75%	max

A

1	2.0	1.5	0.707107	1.0	1.25	1.5	1.75	2.0
---	-----	-----	----------	-----	------	-----	------	-----

2	2.0	3.5	0.707107	3.0	3.25	3.5	3.75	4.0
---	-----	-----	----------	-----	------	-----	------	-----

```
In [107]: df.groupby('A').agg([np.mean, np.std, np.min, np.max])
```

↔

B

	mean	std
--	------	-----

A

1	1.5	0.707107	1	2
---	-----	----------	---	---

2	3.5	0.707107	3	4
---	-----	----------	---	---

### 1.9.2.12 Window Binary Corr/Cov operations return a MultiIndex DataFrame

A binary window operation, like `.corr()` or `.cov()`, when operating on a `.rolling(...)`, `.expanding(...)`, or `.ewm(...)` object, will now return a 2-level MultiIndexed DataFrame rather than a Panel, as Panel is now deprecated, see [here](#). These are equivalent in function, but a MultiIndexed DataFrame enjoys more support in pandas. See the section on *Windowed Binary Operations* for more information. (GH15677)

```
In [108]: np.random.seed(1234)

In [109]: df = pd.DataFrame(np.random.rand(100, 2),
.....:                      columns=pd.Index(['A', 'B'], name='bar'),
.....:                      index=pd.date_range('20160101',
.....:                                          periods=100, freq='D', name='foo'))

In [110]: df.tail()
Out[110]:
bar          A          B
foo
2016-04-05  0.640880  0.126205
2016-04-06  0.171465  0.737086
2016-04-07  0.127029  0.369650
2016-04-08  0.604334  0.103104
2016-04-09  0.802374  0.945553
```

Previous Behavior:

```
In [2]: df.rolling(12).corr()
Out[2]:
<class 'pandas.core.panel.Panel'>
Dimensions: 100 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: 2016-01-01 00:00:00 to 2016-04-09 00:00:00
Major_axis axis: A to B
Minor_axis axis: A to B
```

New Behavior:

```
In [111]: res = df.rolling(12).corr()

In [112]: res.tail()
Out[112]:
bar          A          B
foo    bar
2016-04-07 B   -0.132090  1.000000
2016-04-08 A    1.000000 -0.145775
          B   -0.145775  1.000000
2016-04-09 A    1.000000  0.119645
          B    0.119645  1.000000
```

Retrieving a correlation matrix for a cross-section

```
In [113]: df.rolling(12).corr().loc['2016-04-07']
Out[113]:
bar          A          B
foo    bar
2016-04-07 A    1.000000 -0.13209
          B   -0.13209  1.00000
```

### 1.9.2.13 HDFStore where string comparison

In previous versions most types could be compared to string column in a `HDFStore` usually resulting in an invalid comparison, returning an empty result frame. These comparisons will now raise a `TypeError` (GH15492)

```
In [114]: df = pd.DataFrame({'unparsed_date': ['2014-01-01', '2014-01-01']})

In [115]: df.to_hdf('store.h5', 'key', format='table', data_columns=True)

In [116]: df.dtypes
Out[116]:
unparsed_date    object
dtype: object
```

Previous Behavior:

```
In [4]: pd.read_hdf('store.h5', 'key', where='unparsed_date > ts')
File "<string>", line 1
      (unparsed_date > 1970-01-01 00:00:01.388552400)
                        ^
SyntaxError: invalid token
```

New Behavior:

```
In [18]: ts = pd.Timestamp('2014-01-01')

In [19]: pd.read_hdf('store.h5', 'key', where='unparsed_date > ts')
TypeError: Cannot compare 2014-01-01 00:00:00 of
type <class 'pandas.tslib.Timestamp'> to string column
```

### 1.9.2.14 Index.intersection and inner join now preserve the order of the left Index

`Index.intersection()` now preserves the order of the calling Index (left) instead of the other Index (right) (GH15582). This affects inner joins, `DataFrame.join()` and `merge()`, and the `.align` method.

- `Index.intersection`

```
In [117]: left = pd.Index([2, 1, 0])

In [118]: left
Out[118]: Int64Index([2, 1, 0], dtype='int64')

In [119]: right = pd.Index([1, 2, 3])

In [120]: right
Out[120]: Int64Index([1, 2, 3], dtype='int64')
```

Previous Behavior:

```
In [4]: left.intersection(right)
Out[4]: Int64Index([1, 2], dtype='int64')
```

New Behavior:

```
In [121]: left.intersection(right)
Out[121]: Int64Index([2, 1], dtype='int64')
```

- `DataFrame.join` and `pd.merge`

```
In [122]: left = pd.DataFrame({'a': [20, 10, 0]}, index=[2, 1, 0])

In [123]: left
Out[123]:
   a
2  20
1  10
0   0

In [124]: right = pd.DataFrame({'b': [100, 200, 300]}, index=[1, 2, 3])

In [125]: right
Out[125]:
   b
1  100
2  200
3  300
```

Previous Behavior:

```
In [4]: left.join(right, how='inner')
Out[4]:
   a    b
1  10  100
2  20  200
```

New Behavior:

```
In [126]: left.join(right, how='inner')
Out[126]:
   a    b
2  20  200
1  10  100
```

### 1.9.2.15 Pivot Table always returns a DataFrame

The documentation for `pivot_table()` states that a `DataFrame` is *always* returned. Here a bug is fixed that allowed this to return a `Series` under certain circumstance. (GH4386)

```
In [127]: df = DataFrame({'col1': [3, 4, 5],
.....:                  'col2': ['C', 'D', 'E'],
.....:                  'col3': [1, 3, 9]})
.....:

In [128]: df
Out[128]:
   col1 col2 col3
0     3    C    1
1     4    D    3
2     5    E    9
```

Previous Behavior:

```
In [2]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[2]:
col3  col2
1      C      3
3      D      4
9      E      5
Name: col1, dtype: int64
```

New Behavior:

```
In [129]: df.pivot_table('col1', index=['col3', 'col2'], aggfunc=np.sum)
Out[129]:
           col1
col3 col2
1      C      3
3      D      4
9      E      5
```

### 1.9.2.16 Other API Changes

- `numexpr` version is now required to be  $\geq 2.4.6$  and it will not be used at all if this requisite is not fulfilled ([GH15213](#)).
- `CParserError` has been renamed to `ParserError` in `pd.read_csv()` and will be removed in the future ([GH12665](#))
- `SparseArray.cumsum()` and `SparseSeries.cumsum()` will now always return `SparseArray` and `SparseSeries` respectively ([GH12855](#))
- `DataFrame.applymap()` with an empty `DataFrame` will return a copy of the empty `DataFrame` instead of a `Series` ([GH8222](#))
- `Series.map()` now respects default values of dictionary subclasses with a `__missing__` method, such as `collections.Counter` ([GH15999](#))
- `.loc` has compat with `.ix` for accepting iterators, and `NamedTuples` ([GH15120](#))
- `interpolate()` and `fillna()` will raise a `ValueError` if the `limit` keyword argument is not greater than 0. ([GH9217](#))
- `pd.read_csv()` will now issue a `ParserWarning` whenever there are conflicting values provided by the `dialect` parameter and the user ([GH14898](#))
- `pd.read_csv()` will now raise a `ValueError` for the C engine if the quote character is larger than one byte ([GH11592](#))
- `inplace` arguments now require a boolean value, else a `ValueError` is thrown ([GH14189](#))
- `pandas.api.types.is_datetime64_ns_dtype` will now report `True` on a tz-aware dtype, similar to `pandas.api.types.is_datetime64_any_dtype`
- `DataFrame.asof()` will return a null filled `Series` instead the scalar `NaN` if a match is not found ([GH15118](#))
- Specific support for `copy.copy()` and `copy.deepcopy()` functions on `NDFrame` objects ([GH15444](#))
- `Series.sort_values()` accepts a one element list of bool for consistency with the behavior of `DataFrame.sort_values()` ([GH15604](#))
- `.merge()` and `.join()` on category dtype columns will now preserve the category dtype when possible ([GH10409](#))

- `SparseDataFrame.default_fill_value` will be 0, previously was `nan` in the return from `pd.get_dummies(..., sparse=True)` ([GH15594](#))
- The default behaviour of `Series.str.match` has changed from extracting groups to matching the pattern. The extracting behaviour was deprecated since pandas version 0.13.0 and can be done with the `Series.str.extract` method ([GH5224](#)). As a consequence, the `as_indexer` keyword is ignored (no longer needed to specify the new behaviour) and is deprecated.
- `NaT` will now correctly report `False` for datetimelike boolean operations such as `is_month_start` ([GH15781](#))
- `NaT` will now correctly return `np.nan` for `Timedelta` and `Period` accessors such as `days` and `quarter` ([GH15782](#))
- `NaT` will now returns `NaT` for `tz_localize` and `tz_convert` methods ([GH15830](#))
- `DataFrame` and `Panel` constructors with invalid input will now raise `ValueError` rather than `PandasError`, if called with scalar inputs and not axes ([GH15541](#))
- `DataFrame` and `Panel` constructors with invalid input will now raise `ValueError` rather than `pandas.core.common.PandasError`, if called with scalar inputs and not axes; The exception `PandasError` is removed as well. ([GH15541](#))
- The exception `pandas.core.common.AmbiguousIndexError` is removed as it is not referenced ([GH15541](#))

## 1.9.3 Reorganization of the library: Privacy Changes

### 1.9.3.1 Modules Privacy Has Changed

Some formerly public python/c/c++/cython extension modules have been moved and/or renamed. These are all removed from the public API. Furthermore, the `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are now considered to be PRIVATE. If indicated, a deprecation warning will be issued if you reference theses modules. ([GH12588](#))

Previous Location	New Location	Deprecated
pandas.lib	pandas._libs.lib	X
pandas.tslib	pandas._libs.tslib	X
pandas.computation	pandas.core.computation	X
pandas.msgpack	pandas.io.msgpack	
pandas.index	pandas._libs.index	
pandas.algos	pandas._libs.algos	
pandas.hashtable	pandas._libs.hashtable	
pandas.indexes	pandas.core.indexes	
pandas.json	pandas._libs.json / pandas.io.json	X
pandas.parser	pandas._libs.parsers	X
pandas.formats	pandas.io.formats	
pandas.sparse	pandas.core.sparse	
pandas.tools	pandas.core.reshape	X
pandas.types	pandas.core.dtypes	X
pandas.io.sas.saslib	pandas.io.sas._sas	
pandas._join	pandas._libs.join	
pandas._hash	pandas._libs.hashing	
pandas._period	pandas._libs.period	
pandas._sparse	pandas._libs.sparse	
pandas._testing	pandas._libs.testing	
pandas._window	pandas._libs.window	

Some new subpackages are created with public functionality that is not directly exposed in the top-level namespace: `pandas.errors`, `pandas.plotting` and `pandas.testing` (more details below). Together with `pandas.api.types` and certain functions in the `pandas.io` and `pandas.tseries` submodules, these are now the public subpackages.

Further changes:

- The function `union_categoricals()` is now importable from `pandas.api.types`, formerly from `pandas.types.concat` (GH15998)
- The type import `pandas.tslib.NaTType` is deprecated and can be replaced by using type (`pandas.NaT`) (GH16146)
- The public functions in `pandas.tools.hashing` deprecated from that locations, but are now importable from `pandas.util` (GH16223)
- The modules in `pandas.util`: `decorators`, `print_versions`, `doctools`, `validators`, `depr_module` are now private. Only the functions exposed in `pandas.util` itself are public (GH16223)

### 1.9.3.2 pandas.errors

We are adding a standard public module for all pandas exceptions & warnings `pandas.errors`. (GH14800). Previously these exceptions & warnings could be imported from `pandas.core.common` or `pandas.io.common`. These exceptions and warnings will be removed from the `*.common` locations in a future release. (GH15541)

The following are now part of this API:

```
[ 'DtypeWarning',
  'EmptyDataError',
  'OutOfBoundsDatetime',
  'ParserError',
```

(continues on next page)



(continued from previous page)

```
'ParserWarning',
'PerformanceWarning',
'UnsortedIndexError',
'UnsupportedFunctionCall']
```

### 1.9.3.3 pandas.testing

We are adding a standard module that exposes the public testing functions in `pandas.testing` ([GH9895](#)). Those functions can be used when writing tests for functionality using pandas objects.

The following testing functions are now part of this API:

- `testing.assert_frame_equal()`
- `testing.assert_series_equal()`
- `testing.assert_index_equal()`

### 1.9.3.4 pandas.plotting

A new public `pandas.plotting` module has been added that holds plotting functionality that was previously in either `pandas.tools.plotting` or in the top-level namespace. See the [deprecations sections](#) for more details.

### 1.9.3.5 Other Development Changes

- Building pandas for development now requires `cython >= 0.23` ([GH14831](#))
- Require at least 0.23 version of cython to avoid problems with character encodings ([GH14699](#))
- Switched the test framework to use `pytest` ([GH13097](#))
- Reorganization of tests directory layout ([GH14854](#), [GH15707](#)).

## 1.9.4 Deprecations

### 1.9.4.1 Deprecate `.ix`

The `.ix` indexer is deprecated, in favor of the more strict `.iloc` and `.loc` indexers. `.ix` offers a lot of magic on the inference of what the user wants to do. To wit, `.ix` can decide to index *positionally* OR via *labels*, depending on the data type of the index. This has caused quite a bit of user confusion over the years. The full indexing documentation is [here](#). ([GH14218](#))

The recommended methods of indexing are:

- `.loc` if you want to *label* index
- `.iloc` if you want to *positionally* index.

Using `.ix` will now show a `DeprecationWarning` with a link to some examples of how to convert code [here](#).

```
In [130]: df = pd.DataFrame({'A': [1, 2, 3],
.....:                      'B': [4, 5, 6]},
.....:                      index=list('abc'))
.....:
```

(continues on next page)

(continued from previous page)

```
In [131]: df
Out[131]:
   A  B
a  1  4
b  2  5
c  3  6
```

Previous Behavior, where you wish to get the 0th and the 2nd elements from the index in the 'A' column.

```
In [3]: df.ix[[0, 2], 'A']
Out[3]:
a    1
c    3
Name: A, dtype: int64
```

Using `.loc`. Here we will select the appropriate indexes from the index, then use *label* indexing.

```
In [132]: df.loc[df.index[[0, 2]], 'A']
Out[132]:
a    1
c    3
Name: A, dtype: int64
```

Using `.iloc`. Here we will get the location of the 'A' column, then use *positional* indexing to select things.

```
In [133]: df.iloc[[0, 2], df.columns.get_loc('A')]
Out[133]:
a    1
c    3
Name: A, dtype: int64
```

### 1.9.4.2 Deprecate Panel

Panel is deprecated and will be removed in a future version. The recommended way to represent 3-D data are with a MultiIndex on a DataFrame via the `to_frame()` or with the `xarray` package. Pandas provides a `to_xarray()` method to automate this conversion. For more details see [Deprecate Panel](#) documentation. (GH13563).

```
In [134]: p = tm.makePanel()

In [135]: p
Out[135]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

Convert to a MultiIndex DataFrame

```
In [136]: p.to_frame()
Out[136]:
           ItemA  ItemB  ItemC
major  minor
2000-01-03 A    0.628776 -1.409432  0.209395
```

(continues on next page)

(continued from previous page)

	B	0.988138	-1.347533	-0.896581
	C	-0.938153	1.272395	-0.161137
	D	-0.223019	-0.591863	-1.051539
2000-01-04	A	0.186494	1.422986	-0.592886
	B	-0.072608	0.363565	1.104352
	C	-1.239072	-1.449567	0.889157
	D	2.123692	-0.414505	-0.319561
2000-01-05	A	0.952478	-2.147855	-1.473116
	B	-0.550603	-0.014752	-0.431550
	C	0.139683	-1.195524	0.288377
	D	0.122273	-1.425795	-0.619993

Convert to an xarray DataArray

```
In [137]: p.to_xarray()
Out [137]:
<xarray.DataArray (items: 3, major_axis: 3, minor_axis: 4)>
array([[[ 0.628776,  0.988138, -0.938153, -0.223019],
        [ 0.186494, -0.072608, -1.239072,  2.123692],
        [ 0.952478, -0.550603,  0.139683,  0.122273]],

       [[-1.409432, -1.347533,  1.272395, -0.591863],
        [ 1.422986,  0.363565, -1.449567, -0.414505],
        [-2.147855, -0.014752, -1.195524, -1.425795]],

       [[ 0.209395, -0.896581, -0.161137, -1.051539],
        [-0.592886,  1.104352,  0.889157, -0.319561],
        [-1.473116, -0.43155 ,  0.288377, -0.619993]])
Coordinates:
  * items      (items) object 'ItemA' 'ItemB' 'ItemC'
  * major_axis (major_axis) datetime64[ns] 2000-01-03 2000-01-04 2000-01-05
  * minor_axis (minor_axis) object 'A' 'B' 'C' 'D'
```

### 1.9.4.3 Deprecate `groupby.agg()` with a dictionary when renaming

The `.groupby(...).agg(...)`, `.rolling(...).agg(...)`, and `.resample(...).agg(...)` syntax can accept a variable of inputs, including scalars, list, and a dict of column names to scalars or lists. This provides a useful syntax for constructing multiple (potentially different) aggregations.

However, `.agg(...)` can *also* accept a dict that allows ‘renaming’ of the result columns. This is a complicated and confusing syntax, as well as not consistent between `Series` and `DataFrame`. We are deprecating this ‘renaming’ functionality.

- We are deprecating passing a dict to a grouped/rolled/resampled `Series`. This allowed one to rename the resulting aggregation, but this had a completely different meaning than passing a dictionary to a grouped `DataFrame`, which accepts column-to-aggregations.
- We are deprecating passing a dict-of-dicts to a grouped/rolled/resampled `DataFrame` in a similar manner.

This is an illustrative example:

```
In [138]: df = pd.DataFrame({'A': [1, 1, 1, 2, 2],
.....:                      'B': range(5),
.....:                      'C': range(5)})
.....:
```

(continues on next page)

(continued from previous page)

```
In [139]: df
Out[139]:
```

	A	B	C
0	1	0	0
1	1	1	1
2	1	2	2
3	2	3	3
4	2	4	4

Here is a typical useful syntax for computing different aggregations for different columns. This is a natural, and useful syntax. We aggregate from the dict-to-list by taking the specified columns and applying the list of functions. This returns a `MultiIndex` for the columns (this is *not* deprecated).

```
In [140]: df.groupby('A').agg({'B': 'sum', 'C': 'min'})
Out[140]:
```

	B	C
A		
1	3	0
2	7	3

Here's an example of the first deprecation, passing a dict to a grouped `Series`. This is a combination aggregation & renaming:

```
In [6]: df.groupby('A').B.agg({'foo': 'count'})
FutureWarning: using a dict on a Series for aggregation
is deprecated and will be removed in a future version

Out[6]:
```

	foo
A	
1	3
2	2

You can accomplish the same operation, more idiomatically by:

```
In [141]: df.groupby('A').B.agg(['count']).rename(columns={'count': 'foo'})
Out[141]:
```

	foo
A	
1	3
2	2

Here's an example of the second deprecation, passing a dict-of-dict to a grouped `DataFrame`:

```
In [23]: (df.groupby('A')
          .agg({'B': {'foo': 'sum'}, 'C': {'bar': 'min'}})
          )
FutureWarning: using a dict with renaming is deprecated and
will be removed in a future version

Out[23]:
```

	B	C
	foo	bar
A		
1	3	0
2	7	3

You can accomplish nearly the same by:

```
In [142]: (df.groupby('A')
.....:      .agg({'B': 'sum', 'C': 'min'})
.....:      .rename(columns={'B': 'foo', 'C': 'bar'})
.....: )
Out[142]:
   foo  bar
A
1     3    0
2     7    3
```

#### 1.9.4.4 Deprecate .plotting

The `pandas.tools.plotting` module has been deprecated, in favor of the top level `pandas.plotting` module. All the public plotting functions are now available from `pandas.plotting` ([GH12548](#)).

Furthermore, the top-level `pandas.scatter_matrix` and `pandas.plot_params` are deprecated. Users can import these from `pandas.plotting` as well.

Previous script:

```
pd.tools.plotting.scatter_matrix(df)
pd.scatter_matrix(df)
```

Should be changed to:

```
pd.plotting.scatter_matrix(df)
```

#### 1.9.4.5 Other Deprecations

- `SparseArray.to_dense()` has deprecated the `fill` parameter, as that parameter was not being respected ([GH14647](#))
- `SparseSeries.to_dense()` has deprecated the `sparse_only` parameter ([GH14647](#))
- `Series.repeat()` has deprecated the `reps` parameter in favor of `repeats` ([GH12662](#))
- The `Series` constructor and `.astype` method have deprecated accepting timestamp dtypes without a frequency (e.g. `np.datetime64`) for the `dtype` parameter ([GH15524](#))
- `Index.repeat()` and `MultiIndex.repeat()` have deprecated the `n` parameter in favor of `repeats` ([GH12662](#))
- `Categorical.searchsorted()` and `Series.searchsorted()` have deprecated the `v` parameter in favor of `value` ([GH12662](#))
- `TimedeltaIndex.searchsorted()`, `DatetimeIndex.searchsorted()`, and `PeriodIndex.searchsorted()` have deprecated the `key` parameter in favor of `value` ([GH12662](#))
- `DataFrame.astype()` has deprecated the `raise_on_error` parameter in favor of `errors` ([GH14878](#))
- `Series.sortlevel` and `DataFrame.sortlevel` have been deprecated in favor of `Series.sort_index` and `DataFrame.sort_index` ([GH15099](#))
- importing `concat` from `pandas.tools.merge` has been deprecated in favor of imports from the `pandas` namespace. This should only affect explicit imports ([GH15358](#))

- `Series/DataFrame/Panel consolidate()` been deprecated as a public method. (GH15483)
- The `as_indexer` keyword of `Series.str.match()` has been deprecated (ignored keyword) (GH15257).
- The following top-level pandas functions have been deprecated and will be removed in a future version (GH13790, GH15940)
  - `pd.pnow()`, replaced by `Period.now()`
  - `pd.Term`, is removed, as it is not applicable to user code. Instead use in-line string expressions in the `where` clause when searching in `HDFStore`
  - `pd.Expr`, is removed, as it is not applicable to user code.
  - `pd.match()`, is removed.
  - `pd.groupby()`, replaced by using the `.groupby()` method directly on a `Series/DataFrame`
  - `pd.get_store()`, replaced by a direct call to `pd.HDFStore(...)`
- `is_any_int_dtype`, `is_floating_dtype`, and `is_sequence` are deprecated from `pandas.api.types` (GH16042)

### 1.9.5 Removal of prior version deprecations/changes

- The `pandas.rpy` module is removed. Similar functionality can be accessed through the [rpy2](#) project. See the [R interfacing docs](#) for more details.
- The `pandas.io.ga` module with a `google-analytics` interface is removed (GH11308). Similar functionality can be found in the [Google2Pandas](#) package.
- `pd.to_datetime` and `pd.to_timedelta` have dropped the `coerce` parameter in favor of errors (GH13602)
- `pandas.stats.fama_macbeth`, `pandas.stats.ols`, `pandas.stats.plm` and `pandas.stats.var`, as well as the top-level `pandas.fama_macbeth` and `pandas.ols` routines are removed. Similar functionality can be found in the [statsmodels](#) package. (GH11898)
- The `TimeSeries` and `SparseTimeSeries` classes, aliases of `Series` and `SparseSeries`, are removed (GH10890, GH15098).
- `Series.is_time_series` is dropped in favor of `Series.index.is_all_dates` (GH15098)
- The deprecated `irow`, `icol`, `iget` and `iget_value` methods are removed in favor of `iloc` and `iat` as explained [here](#) (GH10711).
- The deprecated `DataFrame.iterkv()` has been removed in favor of `DataFrame.iteritems()` (GH10711)
- The `Categorical` constructor has dropped the `name` parameter (GH10632)
- `Categorical` has dropped support for `NaN` categories (GH10748)
- The `take_last` parameter has been dropped from `uplicated()`, `drop_duplicates()`, `nlargest()`, and `nsmallest()` methods (GH10236, GH10792, GH10920)
- `Series`, `Index`, and `DataFrame` have dropped the `sort` and `order` methods (GH10726)
- Where clauses in `pytables` are only accepted as strings and expressions types and not other data-types (GH12027)
- `DataFrame` has dropped the `combineAdd` and `combineMult` methods in favor of `add` and `mul` respectively (GH10735)

## 1.9.6 Performance Improvements

- Improved performance of `pd.wide_to_long()` (GH14779)
- Improved performance of `pd.factorize()` by releasing the GIL with `object` dtype when inferred as strings (GH14859, GH16057)
- Improved performance of timeseries plotting with an irregular `DatetimeIndex` (or with `compat_x=True`) (GH15073).
- Improved performance of `groupby().cummin()` and `groupby().cummax()` (GH15048, GH15109, GH15561, GH15635)
- Improved performance and reduced memory when indexing with a `MultiIndex` (GH15245)
- When reading buffer object in `read_sas()` method without specified format, filepath string is inferred rather than buffer object. (GH14947)
- Improved performance of `.rank()` for categorical data (GH15498)
- Improved performance when using `.unstack()` (GH15503)
- Improved performance of merge/join on category columns (GH10409)
- Improved performance of `drop_duplicates()` on bool columns (GH12963)
- Improve performance of `pd.core.groupby.GroupBy.apply` when the applied function used the `.name` attribute of the group `DataFrame` (GH15062).
- Improved performance of `iloc` indexing with a list or array (GH15504).
- Improved performance of `Series.sort_index()` with a monotonic index (GH15694)
- Improved performance in `pd.read_csv()` on some platforms with buffered reads (GH16039)

## 1.9.7 Bug Fixes

### 1.9.7.1 Conversion

- Bug in `Timestamp.replace` now raises `TypeError` when incorrect argument names are given; previously this raised `ValueError` (GH15240)
- Bug in `Timestamp.replace` with `compat` for passing long integers (GH15030)
- Bug in `Timestamp` returning UTC based time/date attributes when a timezone was provided (GH13303, GH6538)
- Bug in `Timestamp` incorrectly localizing timezones during construction (GH11481, GH15777)
- Bug in `TimedeltaIndex` addition where overflow was being allowed without error (GH14816)
- Bug in `TimedeltaIndex` raising a `ValueError` when boolean indexing with `loc` (GH14946)
- Bug in catching an overflow in `Timestamp + Timedelta/Offset` operations (GH15126)
- Bug in `DatetimeIndex.round()` and `Timestamp.round()` floating point accuracy when rounding by milliseconds or less (GH14440, GH15578)
- Bug in `astype()` where `inf` values were incorrectly converted to integers. Now raises error now with `astype()` for `Series` and `DataFrames` (GH14265)
- Bug in `DataFrame(...).apply(to_numeric)` when values are of type `decimal.Decimal`. (GH14827)

- Bug in `describe()` when passing a numpy array which does not contain the median to the `percentiles` keyword argument ([GH14908](#))
- Cleaned up `PeriodIndex` constructor, including raising on floats more consistently ([GH13277](#))
- Bug in using `__deepcopy__` on empty `NDFrame` objects ([GH15370](#))
- Bug in `.replace()` may result in incorrect dtypes. ([GH12747](#), [GH15765](#))
- Bug in `Series.replace` and `DataFrame.replace` which failed on empty replacement dicts ([GH15289](#))
- Bug in `Series.replace` which replaced a numeric by string ([GH15743](#))
- Bug in `Index` construction with `NaN` elements and integer dtype specified ([GH15187](#))
- Bug in `Series` construction with a `datetimez` ([GH14928](#))
- Bug in `Series.dt.round()` inconsistent behaviour on `NaT` 's with different arguments ([GH14940](#))
- Bug in `Series` constructor when both `copy=True` and `dtype` arguments are provided ([GH15125](#))
- Incorrect dtypes `Series` was returned by comparison methods (e.g., `lt`, `gt`, ...) against a constant for an empty `DataFrame` ([GH15077](#))
- Bug in `Series.ffill()` with mixed dtypes containing tz-aware datetimes. ([GH14956](#))
- Bug in `DataFrame.fillna()` where the argument `downcast` was ignored when `fillna` value was of type `dict` ([GH15277](#))
- Bug in `.asfreq()`, where frequency was not set for empty `Series` ([GH14320](#))
- Bug in `DataFrame` construction with nulls and datetimes in a list-like ([GH15869](#))
- Bug in `DataFrame.fillna()` with tz-aware datetimes ([GH15855](#))
- Bug in `is_string_dtype`, `is_timedelta64_ns_dtype`, and `is_string_like_dtype` in which an error was raised when `None` was passed in ([GH15941](#))
- Bug in the return type of `pd.unique` on a `Categorical`, which was returning an `ndarray` and not a `Categorical` ([GH15903](#))
- Bug in `Index.to_series()` where the index was not copied (and so mutating later would change the original), ([GH15949](#))
- Bug in indexing with partial string indexing with a len-1 `DataFrame` ([GH16071](#))
- Bug in `Series` construction where passing invalid dtype didn't raise an error. ([GH15520](#))

### 1.9.7.2 Indexing

- Bug in `Index` power operations with reversed operands ([GH14973](#))
- Bug in `DataFrame.sort_values()` when sorting by multiple columns where one column is of type `int64` and contains `NaT` ([GH14922](#))
- Bug in `DataFrame.reindex()` in which method was ignored when passing columns ([GH14992](#))
- Bug in `DataFrame.loc` with indexing a `MultiIndex` with a `Series` indexer ([GH14730](#), [GH15424](#))
- Bug in `DataFrame.loc` with indexing a `MultiIndex` with a numpy array ([GH15434](#))
- Bug in `Series.asof` which raised if the series contained all `np.nan` ([GH15713](#))
- Bug in `.at` when selecting from a tz-aware column ([GH15822](#))
- Bug in `Series.where()` and `DataFrame.where()` where array-like conditionals were being rejected ([GH15414](#))



- Bug in `Series.where()` where TZ-aware data was converted to float representation ([GH15701](#))
- Bug in `.loc` that would not return the correct dtype for scalar access for a `DataFrame` ([GH11617](#))
- Bug in output formatting of a `MultiIndex` when names are integers ([GH12223](#), [GH15262](#))
- Bug in `Categorical.searchsorted()` where alphabetical instead of the provided categorical order was used ([GH14522](#))
- Bug in `Series.iloc` where a `Categorical` object for list-like indexes input was returned, where a `Series` was expected. ([GH14580](#))
- Bug in `DataFrame.isin` comparing datetimelike to empty frame ([GH15473](#))
- Bug in `.reset_index()` when an all NaN level of a `MultiIndex` would fail ([GH6322](#))
- Bug in `.reset_index()` when raising error for index name already present in `MultiIndex` columns ([GH16120](#))
- Bug in creating a `MultiIndex` with tuples and not passing a list of names; this will now raise `ValueError` ([GH15110](#))
- Bug in the HTML display with with a `MultiIndex` and truncation ([GH14882](#))
- Bug in the display of `.info()` where a qualifier (+) would always be displayed with a `MultiIndex` that contains only non-strings ([GH15245](#))
- Bug in `pd.concat()` where the names of `MultiIndex` of resulting `DataFrame` are not handled correctly when `None` is presented in the names of `MultiIndex` of input `DataFrame` ([GH15787](#))
- Bug in `DataFrame.sort_index()` and `Series.sort_index()` where `na_position` doesn't work with a `MultiIndex` ([GH14784](#), [GH16604](#))
- Bug in `pd.concat()` when combining objects with a `CategoricalIndex` ([GH16111](#))
- Bug in indexing with a scalar and a `CategoricalIndex` ([GH16123](#))

### 1.9.7.3 I/O

- Bug in `pd.to_numeric()` in which float and unsigned integer elements were being improperly casted ([GH14941](#), [GH15005](#))
- Bug in `pd.read_fwf()` where the `skiprows` parameter was not being respected during column width inference ([GH11256](#))
- Bug in `pd.read_csv()` in which the `dialect` parameter was not being verified before processing ([GH14898](#))
- Bug in `pd.read_csv()` in which missing data was being improperly handled with `usecols` ([GH6710](#))
- Bug in `pd.read_csv()` in which a file containing a row with many columns followed by rows with fewer columns would cause a crash ([GH14125](#))
- Bug in `pd.read_csv()` for the C engine where `usecols` were being indexed incorrectly with `parse_dates` ([GH14792](#))
- Bug in `pd.read_csv()` with `parse_dates` when multiline headers are specified ([GH15376](#))
- Bug in `pd.read_csv()` with `float_precision='round_trip'` which caused a segfault when a text entry is parsed ([GH15140](#))
- Bug in `pd.read_csv()` when an index was specified and no values were specified as null values ([GH15835](#))
- Bug in `pd.read_csv()` in which certain invalid file objects caused the Python interpreter to crash ([GH15337](#))

- Bug in `pd.read_csv()` in which invalid values for `nrows` and `chunksize` were allowed ([GH15767](#))
- Bug in `pd.read_csv()` for the Python engine in which unhelpful error messages were being raised when parsing errors occurred ([GH15910](#))
- Bug in `pd.read_csv()` in which the `skipfooter` parameter was not being properly validated ([GH15925](#))
- Bug in `pd.to_csv()` in which there was numeric overflow when a timestamp index was being written ([GH15982](#))
- Bug in `pd.util.hashing.hash_pandas_object()` in which hashing of categoricals depended on the ordering of categories, instead of just their values. ([GH15143](#))
- Bug in `.to_json()` where `lines=True` and contents (keys or values) contain escaped characters ([GH15096](#))
- Bug in `.to_json()` causing single byte ascii characters to be expanded to four byte unicode ([GH15344](#))
- Bug in `.to_json()` for the C engine where rollover was not correctly handled for case where `frac` is odd and `diff` is exactly 0.5 ([GH15716](#), [GH15864](#))
- Bug in `pd.read_json()` for Python 2 where `lines=True` and contents contain non-ascii unicode characters ([GH15132](#))
- Bug in `pd.read_msgpack()` in which Series categoricals were being improperly processed ([GH14901](#))
- Bug in `pd.read_msgpack()` which did not allow loading of a dataframe with an index of type `CategoricalIndex` ([GH15487](#))
- Bug in `pd.read_msgpack()` when deserializing a `CategoricalIndex` ([GH15487](#))
- Bug in `DataFrame.to_records()` with converting a `DatetimeIndex` with a timezone ([GH13937](#))
- Bug in `DataFrame.to_records()` which failed with unicode characters in column names ([GH11879](#))
- Bug in `.to_sql()` when writing a `DataFrame` with numeric index names ([GH15404](#)).
- Bug in `DataFrame.to_html()` with `index=False` and `max_rows` raising in `IndexError` ([GH14998](#))
- Bug in `pd.read_hdf()` passing a `Timestamp` to the `where` parameter with a non date column ([GH15492](#))
- Bug in `DataFrame.to_stata()` and `StataWriter` which produces incorrectly formatted files to be produced for some locales ([GH13856](#))
- Bug in `StataReader` and `StataWriter` which allows invalid encodings ([GH15723](#))
- Bug in the `Series` repr not showing the length when the output was truncated ([GH15962](#)).

#### 1.9.7.4 Plotting

- Bug in `DataFrame.hist` where `plt.tight_layout` caused an `AttributeError` (use `matplotlib >= 2.0.1`) ([GH9351](#))
- Bug in `DataFrame.boxplot` where `fontsize` was not applied to the tick labels on both axes ([GH15108](#))
- Bug in the date and time converters pandas registers with matplotlib not handling multiple dimensions ([GH16026](#))
- Bug in `pd.scatter_matrix()` could accept either `color` or `c`, but not both ([GH14855](#))

### 1.9.7.5 Groupby/Resample/Rolling

- Bug in `.groupby(...).resample()` when passed the `on=` kwarg. (GH15021)
- Properly set `__name__` and `__qualname__` for `Groupby.*` functions (GH14620)
- Bug in `GroupBy.get_group()` failing with a categorical grouper (GH15155)
- Bug in `.groupby(...).rolling(...)` when `on` is specified and using a `DatetimeIndex` (GH15130, GH13966)
- Bug in groupby operations with `timedelta64` when passing `numeric_only=False` (GH5724)
- Bug in `groupby.apply()` coercing object dtypes to numeric types, when not all values were numeric (GH14423, GH15421, GH15670)
- Bug in `resample`, where a non-string `loffset` argument would not be applied when resampling a timeseries (GH13218)
- Bug in `DataFrame.groupby().describe()` when grouping on `Index` containing tuples (GH14848)
- Bug in `groupby().nunique()` with a datetimelike-grouper where bins counts were incorrect (GH13453)
- Bug in `groupby.transform()` that would coerce the resultant dtypes back to the original (GH10972, GH11444)
- Bug in `groupby.agg()` incorrectly localizing timezone on datetime (GH15426, GH10668, GH13046)
- Bug in `.rolling/expanding()` functions where `count()` was not counting `np.Inf`, nor handling object dtypes (GH12541)
- Bug in `.rolling()` where `pd.Timedelta` or `datetime.timedelta` was not accepted as a window argument (GH15440)
- Bug in `Rolling.quantile` function that caused a segmentation fault when called with a quantile value outside of the range `[0, 1]` (GH15463)
- Bug in `DataFrame.resample().median()` if duplicate column names are present (GH14233)

### 1.9.7.6 Sparse

- Bug in `SparseSeries.reindex` on single level with list of length 1 (GH15447)
- Bug in repr-formatting a `SparseDataFrame` after a value was set on (a copy of) one of its series (GH15488)
- Bug in `SparseDataFrame` construction with lists not coercing to dtype (GH15682)
- Bug in sparse array indexing in which indices were not being validated (GH15863)

### 1.9.7.7 Reshaping

- Bug in `pd.merge_asof()` where `left_index` or `right_index` caused a failure when multiple `by` was specified (GH15676)
- Bug in `pd.merge_asof()` where `left_index/right_index` together caused a failure when `tolerance` was specified (GH15135)
- Bug in `DataFrame.pivot_table()` where `dropna=True` would not drop all-NaN columns when the columns was a category dtype (GH15193)
- Bug in `pd.melt()` where passing a tuple value for `value_vars` caused a `TypeError` (GH15348)

- Bug in `pd.pivot_table()` where no error was raised when values argument was not in the columns (GH14938)
- Bug in `pd.concat()` in which concatenating with an empty dataframe with `join='inner'` was being improperly handled (GH15328)
- Bug with `sort=True` in `DataFrame.join` and `pd.merge` when joining on indexes (GH15582)
- Bug in `DataFrame.nsmallest` and `DataFrame.nlargest` where identical values resulted in duplicated rows (GH15297)
- Bug in `pandas.pivot_table()` incorrectly raising `UnicodeError` when passing unicode input for margins keyword (GH13292)

#### 1.9.7.8 Numeric

- Bug in `.rank()` which incorrectly ranks ordered categories (GH15420)
- Bug in `.corr()` and `.cov()` where the column and index were the same object (GH14617)
- Bug in `.mode()` where mode was not returned if was only a single value (GH15714)
- Bug in `pd.cut()` with a single bin on an all 0s array (GH15428)
- Bug in `pd.qcut()` with a single quantile and an array with identical values (GH15431)
- Bug in `pandas.tools.utils.cartesian_product()` with large input can cause overflow on windows (GH15265)
- Bug in `.eval()` which caused multiline evals to fail with local variables not on the first line (GH15342)

#### 1.9.7.9 Other

- Compat with SciPy 0.19.0 for testing on `.interpolate()` (GH15662)
- Compat for 32-bit platforms for `.qcut/cut`; bins will now be `int64` dtype (GH14866)
- Bug in interactions with Qt when a `QtApplication` already exists (GH14372)
- Avoid use of `np.finfo()` during `import pandas` removed to mitigate deadlock on Python GIL misuse (GH14641)

## 1.10 v0.19.2 (December 24, 2016)

This is a minor bug-fix release in the 0.19.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Compatibility with Python 3.6
- Added a [Pandas Cheat Sheet](#). (GH13202).

#### What's new in v0.19.2

- *Enhancements*
- *Performance Improvements*

---

- *Bug Fixes*

---

### 1.10.1 Enhancements

The `pd.merge_asof()`, added in 0.19.0, gained some improvements:

- `pd.merge_asof()` gained `left_index/right_index` and `left_by/right_by` arguments ([GH14253](#))
- `pd.merge_asof()` can take multiple columns in `by` parameter and has specialized dtypes for better performance ([GH13936](#))

### 1.10.2 Performance Improvements

- Performance regression with `PeriodIndex` ([GH14822](#))
- Performance regression in indexing with `getitem` ([GH14930](#))
- Improved performance of `.replace()` ([GH12745](#))
- Improved performance `Series` creation with a datetime index and dictionary data ([GH14894](#))

### 1.10.3 Bug Fixes

- Compat with python 3.6 for pickling of some offsets ([GH14685](#))
- Compat with python 3.6 for some indexing exception types ([GH14684](#), [GH14689](#))
- Compat with python 3.6 for deprecation warnings in the test suite ([GH14681](#))
- Compat with python 3.6 for Timestamp pickles ([GH14689](#))
- Compat with `dateutil==2.6.0`; segfault reported in the testing suite ([GH14621](#))
- Allow nanoseconds in `Timestamp.replace` as a kwarg ([GH14621](#))
- Bug in `pd.read_csv` in which aliasing was being done for `na_values` when passed in as a dictionary ([GH14203](#))
- Bug in `pd.read_csv` in which column indices for a dict-like `na_values` were not being respected ([GH14203](#))
- Bug in `pd.read_csv` where reading files fails, if the number of headers is equal to the number of lines in the file ([GH14515](#))
- Bug in `pd.read_csv` for the Python engine in which an unhelpful error message was being raised when multi-char delimiters were not being respected with quotes ([GH14582](#))
- Fix bugs ([GH14734](#), [GH13654](#)) in `pd.read_sas` and `pandas.io.sas.sas7bdat.SAS7BDATReader` that caused problems when reading a SAS file incrementally.
- Bug in `pd.read_csv` for the Python engine in which an unhelpful error message was being raised when `skipfooter` was not being respected by Python's CSV library ([GH13879](#))
- Bug in `.fillna()` in which timezone aware `datetime64` values were incorrectly rounded ([GH14872](#))
- Bug in `.groupby(..., sort=True)` of a non-lexsorted `MultiIndex` when grouping with multiple levels ([GH14776](#))
- Bug in `pd.cut` with negative values and a single bin ([GH14652](#))

- Bug in `pd.to_numeric` where a 0 was not unsigned on a `downcast='unsigned'` argument (GH14401)
- Bug in plotting regular and irregular timeseries using shared axes (`sharex=True` or `ax.twinx()`) (GH13341, GH14322).
- Bug in not propagating exceptions in parsing invalid datetimes, noted in python 3.6 (GH14561)
- Bug in resampling a `DatetimeIndex` in local TZ, covering a DST change, which would raise `AmbiguousTimeError` (GH14682)
- Bug in indexing that transformed `RecursionError` into `KeyError` or `IndexingError` (GH14554)
- Bug in `HDFStore` when writing a `MultiIndex` when using `data_columns=True` (GH14435)
- Bug in `HDFStore.append()` when writing a `Series` and passing a `min_itemsize` argument containing a value for the index (GH11412)
- Bug when writing to a `HDFStore` in table format with a `min_itemsize` value for the index and without asking to append (GH10381)
- Bug in `Series.groupby().nunique()` raising an `IndexError` for an empty `Series` (GH12553)
- Bug in `DataFrame.nlargest` and `DataFrame.nsmallest` when the index had duplicate values (GH13412)
- Bug in clipboard functions on linux with python2 with unicode and separators (GH13747)
- Bug in clipboard functions on Windows 10 and python 3 (GH14362, GH12807)
- Bug in `.to_clipboard()` and Excel compat (GH12529)
- Bug in `DataFrame.combine_first()` for integer columns (GH14687).
- Bug in `pd.read_csv()` in which the `dtype` parameter was not being respected for empty data (GH14712)
- Bug in `pd.read_csv()` in which the `nrows` parameter was not being respected for large input when using the C engine for parsing (GH7626)
- Bug in `pd.merge_asof()` could not handle timezone-aware `DatetimeIndex` when a tolerance was specified (GH14844)
- Explicit check in `to_stata` and `StataWriter` for out-of-range values when writing doubles (GH14618)
- Bug in `.plot(kind='kde')` which did not drop missing values to generate the KDE Plot, instead generating an empty plot. (GH14821)
- Bug in `unstack()` if called with a list of column(s) as an argument, regardless of the dtypes of all columns, they get coerced to object (GH11847)

## 1.11 v0.19.1 (November 3, 2016)

This is a minor bug-fix release from 0.19.0 and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

### What's new in v0.19.1

- *Performance Improvements*
- *Bug Fixes*

### 1.11.1 Performance Improvements

- Fixed performance regression in factorization of `Period` data ([GH14338](#))
- Fixed performance regression in `Series.asof(where)` when `where` is a scalar ([GH14461](#))
- Improved performance in `DataFrame.asof(where)` when `where` is a scalar ([GH14461](#))
- Improved performance in `.to_json()` when `lines=True` ([GH14408](#))
- Improved performance in certain types of *loc* indexing with a `MultiIndex` ([GH14551](#)).

### 1.11.2 Bug Fixes

- Source installs from PyPI will now again work without `cython` installed, as in previous versions ([GH14204](#))
- Compat with Cython 0.25 for building ([GH14496](#))
- Fixed regression where user-provided file handles were closed in `read_csv(c engine)` ([GH14418](#)).
- Fixed regression in `DataFrame.quantile` when missing values were present in some columns ([GH14357](#)).
- Fixed regression in `Index.difference` where the `freq` of a `DatetimeIndex` was incorrectly set ([GH14323](#))
- Added back `pandas.core.common.array_equivalent` with a deprecation warning ([GH14555](#)).
- Bug in `pd.read_csv` for the C engine in which quotation marks were improperly parsed in skipped rows ([GH14459](#))
- Bug in `pd.read_csv` for Python 2.x in which Unicode quote characters were no longer being respected ([GH14477](#))
- Fixed regression in `Index.append` when categorical indices were appended ([GH14545](#)).
- Fixed regression in `pd.DataFrame` where constructor fails when given dict with `None` value ([GH14381](#))
- Fixed regression in `DatetimeIndex._maybe_cast_slice_bound` when index is empty ([GH14354](#)).
- Bug in localizing an ambiguous timezone when a boolean is passed ([GH14402](#))
- Bug in `TimedeltaIndex` addition with a `Datetime`-like object where addition overflow in the negative direction was not being caught ([GH14068](#), [GH14453](#))
- Bug in string indexing against data with object `Index` may raise `AttributeError` ([GH14424](#))
- Correctly raise `ValueError` on empty input to `pd.eval()` and `df.query()` ([GH13139](#))
- Bug in `RangeIndex.intersection` when result is a empty set ([GH14364](#)).
- Bug in groupby-transform broadcasting that could cause incorrect dtype coercion ([GH14457](#))
- Bug in `Series.__setitem__` which allowed mutating read-only arrays ([GH14359](#)).
- Bug in `DataFrame.insert` where multiple calls with duplicate columns can fail ([GH14291](#))
- `pd.merge()` will raise `ValueError` with non-boolean parameters in passed boolean type arguments ([GH14434](#))
- Bug in `Timestamp` where dates very near the minimum (1677-09) could underflow on creation ([GH14415](#))
- Bug in `pd.concat` where names of the keys were not propagated to the resulting `MultiIndex` ([GH14252](#))
- Bug in `pd.concat` where `axis` cannot take string parameters `'rows'` or `'columns'` ([GH14369](#))

- Bug in `pd.concat` with dataframes heterogeneous in length and tuple keys ([GH14438](#))
- Bug in `MultiIndex.set_levels` where illegal level values were still set after raising an error ([GH13754](#))
- Bug in `DataFrame.to_json` where `lines=True` and a value contained a `}` character ([GH14391](#))
- Bug in `df.groupby` causing an `AttributeError` when grouping a single index frame by a column and the index level ([GH14327](#))
- Bug in `df.groupby` where `TypeError` raised when `pd.Grouper(key=...)` is passed in a list ([GH14334](#))
- Bug in `pd.pivot_table` may raise `TypeError` or `ValueError` when index or columns is not scalar and values is not specified ([GH14380](#))

## 1.12 v0.19.0 (October 2, 2016)

This is a major release from 0.18.1 and includes number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- `merge_asof()` for asof-style time-series joining, see [here](#)
- `.rolling()` is now time-series aware, see [here](#)
- `read_csv()` now supports parsing Categorical data, see [here](#)
- A function `union_categorical()` has been added for combining categoricals, see [here](#)
- `PeriodIndex` now has its own `period` dtype, and changed to be more consistent with other `Index` classes. See [here](#)
- Sparse data structures gained enhanced support of `int` and `bool` dtypes, see [here](#)
- Comparison operations with `Series` no longer ignores the index, see [here](#) for an overview of the API changes.
- Introduction of a pandas development API for utility functions, see [here](#).
- Deprecation of `Panel4D` and `PanelND`. We recommend to represent these types of n-dimensional data with the `xarray` package.
- Removal of the previously deprecated modules `pandas.io.data`, `pandas.io.wb`, `pandas.tools.rplot`.

**Warning:** pandas >= 0.19.0 will no longer silence numpy ufunc warnings upon import, see [here](#).

### What's new in v0.19.0

- *New features*
  - *`merge_asof` for asof-style time-series joining*
  - *`.rolling()` is now time-series aware*
  - *`read_csv` has improved support for duplicate column names*
  - *`read_csv` supports parsing Categorical directly*



- *Categorical Concatenation*
- *Semi-Month Offsets*
- *New Index methods*
- *Google BigQuery Enhancements*
- *Fine-grained numpy errstate*
- *get\_dummies now returns integer dtypes*
- *Downcast values to smallest possible dtype in to\_numeric*
- *pandas development API*
- *Other enhancements*
- *API changes*
  - *Series.tolist() will now return Python types*
  - *Series operators for different indexes*
    - \* *Arithmetic operators*
    - \* *Comparison operators*
    - \* *Logical operators*
    - \* *Flexible comparison methods*
  - *Series type promotion on assignment*
  - *.to\_datetime() changes*
  - *Merging changes*
  - *.describe() changes*
  - *Period changes*
    - \* *PeriodIndex now has period dtype*
    - \* *Period('NaT') now returns pd.NaT*
    - \* *PeriodIndex.values now returns array of Period object*
  - *Index +/- no longer used for set operations*
  - *Index.difference and .symmetric\_difference changes*
  - *Index.unique consistently returns Index*
  - *MultiIndex constructors, groupby and set\_index preserve categorical dtypes*
  - *read\_csv will progressively enumerate chunks*
  - *Sparse Changes*
    - \* *int64 and bool support enhancements*
    - \* *Operators now preserve dtypes*
    - \* *Other sparse fixes*
  - *Indexer dtype changes*
  - *Other API Changes*

- *Deprecations*
- *Removal of prior version deprecations/changes*
- *Performance Improvements*
- *Bug Fixes*

## 1.12.1 New features

### 1.12.1.1 `merge_asof` for asof-style time-series joining

A long-time requested feature has been added through the `merge_asof()` function, to support asof style joining of time-series (GH1870, GH13695, GH13709, GH13902). Full documentation is [here](#).

The `merge_asof()` performs an asof merge, which is similar to a left-join except that we match on nearest key rather than equal keys.

```
In [1]: left = pd.DataFrame({'a': [1, 5, 10],
...:                        'left_val': ['a', 'b', 'c']})
...:
...:

In [2]: right = pd.DataFrame({'a': [1, 2, 3, 6, 7],
...:                          'right_val': [1, 2, 3, 6, 7]})
...:
...:

In [3]: left
Out[3]:
   a left_val
0  1         a
1  5         b
2 10         c

In [4]: right
Out[4]:
   a right_val
0  1          1
1  2          2
2  3          3
3  6          6
4  7          7
```

We typically want to match exactly when possible, and use the most recent value otherwise.

```
In [5]: pd.merge_asof(left, right, on='a')
Out[5]:
   a left_val right_val
0  1         a         1
1  5         b         3
2 10         c         7
```

We can also match rows ONLY with prior data, and not an exact match.

```
In [6]: pd.merge_asof(left, right, on='a', allow_exact_matches=False)
Out[6]:
   a left_val right_val
```

(continues on next page)

0	1	a	NaN
1	5	b	3.0
2	10	c	7.0

```
In [7]: trades = pd.DataFrame({
...:     'time': pd.to_datetime(['20160525 13:30:00.023',
...:                               '20160525 13:30:00.038',
...:                               '20160525 13:30:00.048',
...:                               '20160525 13:30:00.048',
...:                               '20160525 13:30:00.048']),
...:     'ticker': ['MSFT', 'MSFT',
...:                 'GOOG', 'GOOG', 'AAPL'],
...:     'price': [51.95, 51.95,
...:               720.77, 720.92, 98.00],
...:     'quantity': [75, 155,
...:                  100, 100, 100]},
...:     columns=['time', 'ticker', 'price', 'quantity'])
...:
```

```
In [8]: quotes = pd.DataFrame({
...:     'time': pd.to_datetime(['20160525 13:30:00.023',
...:                               '20160525 13:30:00.023',
...:                               '20160525 13:30:00.030',
...:                               '20160525 13:30:00.041',
...:                               '20160525 13:30:00.048',
...:                               '20160525 13:30:00.049',
...:                               '20160525 13:30:00.072',
...:                               '20160525 13:30:00.075']),
...:     'ticker': ['GOOG', 'MSFT', 'MSFT',
...:                 'MSFT', 'GOOG', 'AAPL', 'GOOG',
...:                 'MSFT'],
...:     'bid': [720.50, 51.95, 51.97, 51.99,
...:             720.50, 97.99, 720.50, 52.01],
...:     'ask': [720.93, 51.96, 51.98, 52.00,
...:             720.93, 98.01, 720.88, 52.03]},
...:     columns=['time', 'ticker', 'bid', 'ask'])
...:
```

```
In [9]: trades
Out[9]:
```

		time	ticker	price	quantity
0	2016-05-25	13:30:00.023	MSFT	51.95	75
1	2016-05-25	13:30:00.038	MSFT	51.95	155
2	2016-05-25	13:30:00.048	GOOG	720.77	100
3	2016-05-25	13:30:00.048	GOOG	720.92	100
4	2016-05-25	13:30:00.048	AAPL	98.00	100

```
In [10]: quotes
\\/////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↪
```

			time ticker		bid	ask
0	2016-05-25	13:30:00.023	GOOG		720.50	720.93
1	2016-05-25	13:30:00.023	MSFT		51.95	51.96

(continued from previous page)

2	2016-05-25 13:30:00.030	MSFT	51.97	51.98
3	2016-05-25 13:30:00.041	MSFT	51.99	52.00
4	2016-05-25 13:30:00.048	GOOG	720.50	720.93
5	2016-05-25 13:30:00.049	AAPL	97.99	98.01
6	2016-05-25 13:30:00.072	GOOG	720.50	720.88
7	2016-05-25 13:30:00.075	MSFT	52.01	52.03

An asof merge joins on the `on`, typically a datetimelike field, which is ordered, and in this case we are using a grouper in the `by` field. This is like a left-outer join, except that forward filling happens automatically taking the most recent non-NaN value.

```
In [11]: pd.merge_asof(trades, quotes,
.....:                  on='time',
.....:                  by='ticker')
.....:
Out [11]:
```

	time	ticker	price	quantity	bid	ask
0	2016-05-25 13:30:00.023	MSFT	51.95	75	51.95	51.96
1	2016-05-25 13:30:00.038	MSFT	51.95	155	51.97	51.98
2	2016-05-25 13:30:00.048	GOOG	720.77	100	720.50	720.93
3	2016-05-25 13:30:00.048	GOOG	720.92	100	720.50	720.93
4	2016-05-25 13:30:00.048	AAPL	98.00	100	NaN	NaN

This returns a merged DataFrame with the entries in the same order as the original left passed DataFrame (`trades` in this case), with the fields of the quotes merged.

### 1.12.1.2 `.rolling()` is now time-series aware

`.rolling()` objects are now time-series aware and can accept a time-series offset (or convertible) for the window argument (GH13327, GH12995). See the full documentation [here](#).

```
In [12]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
.....:                      index=pd.date_range('20130101 09:00:00', periods=5, freq=
↪ 's'))
.....:

In [13]: dft
Out [13]:
```

	B
2013-01-01 09:00:00	0.0
2013-01-01 09:00:01	1.0
2013-01-01 09:00:02	2.0
2013-01-01 09:00:03	NaN
2013-01-01 09:00:04	4.0

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [14]: dft.rolling(2).sum()
Out [14]:
```

	B
2013-01-01 09:00:00	NaN
2013-01-01 09:00:01	1.0
2013-01-01 09:00:02	3.0
2013-01-01 09:00:03	NaN
2013-01-01 09:00:04	NaN

(continues on next page)

(continued from previous page)

```
In [15]: dft.rolling(2, min_periods=1).sum()
```

```

////////////////////////////////////
↪
      B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0

```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```
In [16]: dft.rolling('2s').sum()
```

```
Out [16]:
```

```

      B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  3.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:04  4.0

```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```
In [17]: dft = DataFrame({'B': [0, 1, 2, np.nan, 4]},
.....:                  index = pd.Index([pd.Timestamp('20130101 09:00:00'),
.....:                                     pd.Timestamp('20130101 09:00:02'),
.....:                                     pd.Timestamp('20130101 09:00:03'),
.....:                                     pd.Timestamp('20130101 09:00:05'),
.....:                                     pd.Timestamp('20130101 09:00:06')],
.....:                                     name='foo'))
```

```
In [18]: dft
```

```
Out [18]:
```

```

      B
foo
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0

```

```
In [19]: dft.rolling(2).sum()
```

```

////////////////////////////////////
↪
      B
foo
2013-01-01 09:00:00  NaN
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  NaN

```

Using the time-specification generates variable windows for this sparse data.

```
In [20]: dft.rolling('2s').sum()
Out[20]:
```

		B
foo		
2013-01-01	09:00:00	0.0
2013-01-01	09:00:02	1.0
2013-01-01	09:00:03	3.0
2013-01-01	09:00:05	NaN
2013-01-01	09:00:06	4.0

Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a `DataFrame`.

```
In [21]: dft = dft.reset_index()
```

```
In [22]: dft
```

Out [22] :

			foo	B
0	2013-01-01	09:00:00	0.0	
1	2013-01-01	09:00:02	1.0	
2	2013-01-01	09:00:03	2.0	
3	2013-01-01	09:00:05	NaN	
4	2013-01-01	09:00:06	4.0	

```
In [23]: dft.rolling('2s', on='foo').sum()
```

---

```

      foo      B
0 2013-01-01 09:00:00 0.0
1 2013-01-01 09:00:02 1.0
2 2013-01-01 09:00:03 3.0
3 2013-01-01 09:00:05 NaN
4 2013-01-01 09:00:06 4.0

```

### 1.12.1.3 read\_csv has improved support for duplicate column names

*Duplicate column names* are now supported in `read_csv()` whether they are in the file or passed in as the `names` parameter ([GH7160](#), [GH9424](#))

```
In [24]: data = '0,1,2\n3,4,5'
```

```
In [25]: names = ['a', 'b', 'a']
```

**Previous behavior:**

```
In [2]: pd.read_csv(StringIO(data), names=names)
```

Out [2] :

	a	b	a
0	2	1	2
1	5	4	5

The first a column contained the same data as the second a column, when it should have contained the values [0, 3].

**New behavior:**

```
In [26]: pd.read_csv(StringIO(data), names=names)
Out[26]:
```

	a	b	a.1
0	0	1	2
1	3	4	5

#### 1.12.1.4 read\_csv supports parsing Categorical directly

The `read_csv()` function now supports parsing a Categorical column when specified as a dtype ([GH10153](#)). Depending on the structure of the data, this can result in a faster parse time and lower memory usage compared to converting to Categorical after parsing. See the [io docs here](#).

```
In [27]: data = 'coll,col2,col3\na,b,1\na,b,2\nc,d,3'

In [28]: pd.read_csv(StringIO(data))
Out[28]:
```

	coll	col2	col3
0	a	b	1
1	a	b	2
2	c	d	3

```
In [29]: pd.read_csv(StringIO(data)).dtypes
Out[29]:
coll      object
col2      object
col3      int64
dtype: object

In [30]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[30]:
coll      category
col2      category
col3      category
dtype: object
```

Individual columns can be parsed as a Categorical using a dict specification

```
In [31]: pd.read_csv(StringIO(data), dtype={'coll': 'category'}).dtypes
Out[31]:
```

	coll	col2	col3
0	a	b	1
1	a	b	2
2	c	d	3

```
dtype: object
```

**Note:** The resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the `to_numeric()` function, or as appropriate, another converter such as `to_datetime()`.

```
In [32]: df = pd.read_csv(StringIO(data), dtype='category')

In [33]: df.dtypes
Out[33]:
coll      category
```

(continues on next page)

(continued from previous page)

```

col2    category
col3    category
dtype: object

In [34]: df['col3']
Out[34]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, object): [1, 2, 3]

In [35]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [36]: df['col3']
Out[36]:
0    1
1    2
2    3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]

```

### 1.12.1.5 Categorical Concatenation

- A function `union_categoricals()` has been added for combining categoricals, see *Unioning Categoricals* (GH13361, GH13763, GH13846, GH14173)

```

In [37]: from pandas.api.types import union_categoricals

In [38]: a = pd.Categorical(["b", "c"])

In [39]: b = pd.Categorical(["a", "b"])

In [40]: union_categoricals([a, b])
Out[40]:
[b, c, a, b]
Categories (3, object): [b, c, a]

```

- `concat` and `append` now can concat category dtypes with different categories as object dtype (GH13524)

```

In [41]: s1 = pd.Series(['a', 'b'], dtype='category')

In [42]: s2 = pd.Series(['b', 'c'], dtype='category')

```

#### Previous behavior:

```

In [1]: pd.concat([s1, s2])
ValueError: incompatible categories in categorical concat

```

#### New behavior:

```

In [43]: pd.concat([s1, s2])
Out[43]:

```

(continues on next page)



(continued from previous page)

```

0      a
1      b
0      b
1      c
dtype: object

```

### 1.12.1.6 Semi-Month Offsets

Pandas has gained new frequency offsets, `SemiMonthEnd` ('SM') and `SemiMonthBegin` ('SMS'). These provide date offsets anchored (by default) to the 15th and end of month, and 15th and 1st of month respectively. (GH1543)

```
In [44]: from pandas.tseries.offsets import SemiMonthEnd, SemiMonthBegin
```

#### SemiMonthEnd:

```

In [45]: Timestamp('2016-01-01') + SemiMonthEnd()
Out[45]: Timestamp('2016-01-15 00:00:00')

In [46]: pd.date_range('2015-01-01', freq='SM', periods=4)
Out[46]: DatetimeIndex(['2015-01-15', '2015-01-31', '2015-02-15', '2015-02-28'], dtype='datetime64[ns]', freq='SM-15')

```

#### SemiMonthBegin:

```

In [47]: Timestamp('2016-01-01') + SemiMonthBegin()
Out[47]: Timestamp('2016-01-15 00:00:00')

In [48]: pd.date_range('2015-01-01', freq='SMS', periods=4)
Out[48]: DatetimeIndex(['2015-01-01', '2015-01-15', '2015-02-01', '2015-02-15'], dtype='datetime64[ns]', freq='SMS-15')

```

Using the anchoring suffix, you can also specify the day of month to use instead of the 15th.

```

In [49]: pd.date_range('2015-01-01', freq='SMS-16', periods=4)
Out[49]: DatetimeIndex(['2015-01-01', '2015-01-16', '2015-02-01', '2015-02-16'], dtype='datetime64[ns]', freq='SMS-16')

In [50]: pd.date_range('2015-01-01', freq='SM-14', periods=4)
Out[50]: DatetimeIndex(['2015-01-14', '2015-01-31', '2015-02-14', '2015-02-28'], dtype='datetime64[ns]', freq='SM-14')

```

### 1.12.1.7 New Index methods

The following methods and options are added to `Index`, to be more consistent with the `Series` and `DataFrame` API.

`Index` now supports the `.where()` function for same shape indexing (GH13170)

```

In [51]: idx = pd.Index(['a', 'b', 'c'])

In [52]: idx.where([True, False, True])
Out[52]: Index(['a', nan, 'c'], dtype='object')

```

Index now supports `.dropna()` to exclude missing values ([GH6194](#))

```
In [53]: idx = pd.Index([1, 2, np.nan, 4])

In [54]: idx.dropna()
Out[54]: Float64Index([1.0, 2.0, 4.0], dtype='float64')
```

For `MultiIndex`, values are dropped if any level is missing by default. Specifying `how='all'` only drops values where all levels are missing.

[illegible]

Index now supports `.str.extractall()` which returns a DataFrame, see the [docs here](#) (GH10008, GH13156)

```
In [59]: idx = pd.Index(["a1a2", "b1", "c1"])

In [60]: idx.str.extractall("[ab](?P<digit>\\d)")
Out[60]:
      digit
match
0 0      1
  1      2
1 0      1
```

`Index.astype()` now accepts an optional boolean argument `copy`, which allows optional copying if the requirements on `dtype` are satisfied ([GH13209](#))

### 1.12.1.8 Google BigQuery Enhancements

- The `read_gbq()` method has gained the `dialect` argument to allow users to specify whether to use BigQuery's legacy SQL or BigQuery's standard SQL. See the [docs](#) for more details ([GH13615](#)).
- The `to_gbq()` method now allows the DataFrame column order to differ from the destination table schema ([GH11359](#)).

### 1.12.1.9 Fine-grained numpy errstate

Previous versions of pandas would permanently silence numpy’s ufunc error handling when pandas was imported. Pandas did this in order to silence the warnings that would arise from using numpy ufuncs on missing data, which are usually represented as NaN s. Unfortunately, this silenced legitimate warnings arising in non-pandas code in the application. Starting with 0.19.0, pandas will use the `numpy.errstate` context manager to silence these warnings in a more fine-grained manner, only around where these operations are actually used in the pandas codebase. ([GH13109](#), [GH13145](#))

After upgrading pandas, you may see *new* `RuntimeWarnings` being issued from your code. These are likely legitimate, and the underlying cause likely existed in the code when using previous versions of pandas that simply silenced the warning. Use `numpy.errstate` around the source of the `RuntimeWarning` to control how these conditions are handled.

### 1.12.1.10 `get_dummies` now returns integer dtypes

The `pd.get_dummies` function now returns dummy-encoded columns as small integers, rather than floats ([GH8725](#)). This should provide an improved memory footprint.

**Previous behavior:**

```
In [1]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes

Out[1]:
a    float64
b    float64
c    float64
dtype: object
```

**New behavior:**

```
In [61]: pd.get_dummies(['a', 'b', 'a', 'c']).dtypes
Out[61]:
a    uint8
b    uint8
c    uint8
dtype: object
```

### 1.12.1.11 Downcast values to smallest possible dtype in `to_numeric`

`pd.to_numeric()` now accepts a `downcast` parameter, which will downcast the data if possible to smallest specified numerical dtype ([GH13352](#))

```
In [62]: s = ['1', 2, 3]

In [63]: pd.to_numeric(s, downcast='unsigned')
Out[63]: array([1, 2, 3], dtype=uint8)

In [64]: pd.to_numeric(s, downcast='integer')
Out[64]: array([1, 2, 3], dtype=int8)
```

### 1.12.1.12 pandas development API

As part of making pandas API more uniform and accessible in the future, we have created a standard sub-package of pandas, `pandas.api` to hold public API's. We are starting by exposing type introspection functions in `pandas.api.types`. More sub-packages and officially sanctioned API's will be published in future versions of pandas (GH13147, GH13634)

The following are now part of this API:

```
In [65]: import pprint

In [66]: from pandas.api import types

In [67]: funcs = [ f for f in dir(types) if not f.startswith('_') ]

In [68]: pprint.pprint(funcs)
['CategoricalDtype',
 'DatetimeTZDtype',
 'IntervalDtype',
 'PeriodDtype',
 'infer_dtype',
 'is_any_int_dtype',
 'is_array_like',
 'is_bool',
 'is_bool_dtype',
 'is_categorical',
 'is_categorical_dtype',
 'is_complex',
 'is_complex_dtype',
 'is_datetime64_any_dtype',
 'is_datetime64_dtype',
 'is_datetime64_ns_dtype',
 'is_datetime64tz_dtype',
 'is_datetimetz',
 'is_dict_like',
 'is_dtype_equal',
 'is_extension_type',
 'is_file_like',
 'is_float',
 'is_float_dtype',
 'is_floating_dtype',
 'is_hashable',
 'is_int64_dtype',
 'is_integer',
 'is_integer_dtype',
 'is_interval',
 'is_interval_dtype',
 'is_iterator',
 'is_list_like',
 'is_named_tuple',
 'is_number',
 'is_numeric_dtype',
 'is_object_dtype',
 'is_period',
 'is_period_dtype',
 'is_re',
 'is_re_compilable',
 'is_scalar',
```

(continues on next page)

(continued from previous page)

```
'is_sequence',
'is_signed_integer_dtype',
'is_sparse',
'is_string_dtype',
'is_timedelta64_dtype',
'is_timedelta64_ns_dtype',
'is_unsigned_integer_dtype',
'pandas_dtype',
'union_categoricals']
```

**Note:** Calling these functions from the internal module `pandas.core.common` will now show a `DeprecationWarning` ([GH13990](#))

### 1.12.1.13 Other enhancements

- `Timestamp` can now accept positional and keyword parameters similar to `datetime.datetime()` ([GH10758](#), [GH11630](#))

```
In [69]: pd.Timestamp(2012, 1, 1)
Out[69]: Timestamp('2012-01-01 00:00:00')

In [70]: pd.Timestamp(year=2012, month=1, day=1, hour=8, minute=30)
Out[70]: Timestamp('2012-01-01 08:30:00
↪')
```

- The `.resample()` function now accepts a `on=` or `level=` parameter for resampling on a datetimelike column or MultiIndex level ([GH13500](#))

```
In [71]: df = pd.DataFrame({'date': pd.date_range('2015-01-01', freq='W',
↪periods=5),
.....:                    'a': np.arange(5)},
.....:                    index=pd.MultiIndex.from_arrays([
.....:                        [1,2,3,4,5],
.....:                        pd.date_range('2015-01-01', freq='W',
↪periods=5)],
.....:                    names=['v','d']))

In [72]: df
Out[72]:
```

		date	a
v	d		
1	2015-01-04	2015-01-04	0
2	2015-01-11	2015-01-11	1
3	2015-01-18	2015-01-18	2
4	2015-01-25	2015-01-25	3
5	2015-02-01	2015-02-01	4

```
In [73]: df.resample('M', on='date').sum()
Out[73]:
```

	a
date	

(continues on next page)

(continued from previous page)

```
2015-01-31    6
2015-02-28    4
```

```
In [74]: df.resample('M', level='d').sum()
```

```

////////////////////////////////////
↪
          a
d
2015-01-31    6
2015-02-28    4

```

- The `.get_credentials()` method of `GbqConnector` can now first try to fetch the [application default credentials](#). See the docs for more details ([GH13577](#)).
- The `.tz_localize()` method of `DatetimeIndex` and `Timestamp` has gained the `errors` keyword, so you can potentially coerce nonexistent timestamps to `NaT`. The default behavior remains to raising a `NonExistentTimeError` ([GH13057](#)).
- `.to_hdf/read_hdf()` now accept path objects (e.g. `pathlib.Path`, `py.path.local`) for the file path ([GH11773](#)).
- The `pd.read_csv()` with `engine='python'` has gained support for the `decimal` ([GH12933](#)), `na_filter` ([GH13321](#)) and the `memory_map` option ([GH13381](#)).
- Consistent with the Python API, `pd.read_csv()` will now interpret `+inf` as positive infinity ([GH13274](#)).
- The `pd.read_html()` has gained support for the `na_values`, `converters`, `keep_default_na` options ([GH13461](#)).
- `Categorical.astype()` now accepts an optional boolean argument `copy`, effective when `dtype` is categorical ([GH13209](#)).
- `DataFrame` has gained the `.asof()` method to return the last non-`NaN` values according to the selected subset ([GH13358](#)).
- The `DataFrame` constructor will now respect key ordering if a list of `OrderedDict` objects are passed in ([GH13304](#)).
- `pd.read_html()` has gained support for the `decimal` option ([GH12907](#)).
- `Series` has gained the properties `.is_monotonic`, `.is_monotonic_increasing`, `.is_monotonic_decreasing`, similar to `Index` ([GH13336](#)).
- `DataFrame.to_sql()` now allows a single value as the SQL type for all columns ([GH11886](#)).
- `Series.append` now supports the `ignore_index` option ([GH13677](#)).
- `.to_stata()` and `StataWriter` can now write variable labels to Stata dta files using a dictionary to make column names to labels ([GH13535](#), [GH13536](#)).
- `.to_stata()` and `StataWriter` will automatically convert `datetime64[ns]` columns to Stata format `%tc`, rather than raising a `ValueError` ([GH12259](#)).
- `read_stata()` and `StataReader` raise with a more explicit error message when reading Stata files with repeated value labels when `convert_categoricals=True` ([GH13923](#)).
- `DataFrame.style` will now render sparsified `MultiIndexes` ([GH11655](#)).
- `DataFrame.style` will now show column level names (e.g. `DataFrame.columns.names`) ([GH13775](#)).
- `DataFrame` has gained support to re-order the columns based on the values in a row using `df.sort_values(by='...', axis=1)` ([GH10806](#)).

```

In [75]: df = pd.DataFrame({'A': [2, 7], 'B': [3, 5], 'C': [4, 8]},
.....:                    index=['row1', 'row2'])
.....:

In [76]: df
Out[76]:
   A  B  C
row1 2  3  4
row2 7  5  8

In [77]: df.sort_values(by='row2', axis=1)
Out[77]:
   B  A  C
row1 3  2  4
row2 5  7  8

```

- Added documentation to *I/O* regarding the perils of reading in columns with mixed dtypes and how to handle it (GH13746)
- `to_html()` now has a `border` argument to control the value in the opening `<table>` tag. The default is the value of the `html.border` option, which defaults to 1. This also affects the notebook HTML repr, but since Jupyter's CSS includes a border-width attribute, the visual effect is the same. (GH11563).
- Raise `ImportError` in the `sql` functions when `sqlalchemy` is not installed and a connection string is used (GH11920).
- Compatibility with matplotlib 2.0. Older versions of pandas should also work with matplotlib 2.0 (GH13333)
- `Timestamp`, `Period`, `DatetimeIndex`, `PeriodIndex` and `.dt` accessor have gained a `.is_leap_year` property to check whether the date belongs to a leap year. (GH13727)
- `astype()` will now accept a dict of column name to data types mapping as the `dtype` argument. (GH12086)
- The `pd.read_json` and `DataFrame.to_json` has gained support for reading and writing json lines with `lines` option see *Line delimited json* (GH9180)
- `read_excel()` now supports the `true_values` and `false_values` keyword arguments (GH13347)
- `groupby()` will now accept a scalar and a single-element list for specifying `level` on a non-`MultiIndex` grouper. (GH13907)
- Non-convertible dates in an excel date column will be returned without conversion and the column will be `object` dtype, rather than raising an exception (GH10001).
- `pd.Timedelta(None)` is now accepted and will return `NaT`, mirroring `pd.Timestamp` (GH13687)
- `pd.read_stata()` can now handle some format 111 files, which are produced by SAS when generating Stata dta files (GH11526)
- `Series` and `Index` now support `divmod` which will return a tuple of series or indices. This behaves like a standard binary operator with regards to broadcasting rules (GH14208).

## 1.12.2 API changes

### 1.12.2.1 `Series.tolist()` will now return Python types

`Series.tolist()` will now return Python types in the output, mimicking `NumPy.tolist()` behavior (GH10904)

```
In [78]: s = pd.Series([1,2,3])
```

**Previous behavior:**

```
In [7]: type(s.tolist()[0])
Out[7]:
<class 'numpy.int64'>
```

**New behavior:**

```
In [79]: type(s.tolist()[0])
Out[79]: int
```

### 1.12.2.2 Series operators for different indexes

Following Series operators have been changed to make all operators consistent, including DataFrame ([GH1134](#), [GH4581](#), [GH13538](#))

- Series comparison operators now raise `ValueError` when index are different.
- Series logical operators align both index of left and right hand side.

**Warning:** Until 0.18.1, comparing Series with the same length, would succeed even if the `.index` are different (the result ignores `.index`). As of 0.19.0, this will raises `ValueError` to be more strict. This section also describes how to keep previous behavior or align different indexes, using the flexible comparison methods like `.eq`.

As a result, Series and DataFrame operators behave as below:

#### Arithmetic operators

Arithmetic operators align both index (no changes).

```
In [80]: s1 = pd.Series([1, 2, 3], index=list('ABC'))
In [81]: s2 = pd.Series([2, 2, 2], index=list('ABD'))
In [82]: s1 + s2
Out[82]:
A    3.0
B    4.0
C     NaN
D     NaN
dtype: float64

In [83]: df1 = pd.DataFrame([1, 2, 3], index=list('ABC'))
In [84]: df2 = pd.DataFrame([2, 2, 2], index=list('ABD'))
In [85]: df1 + df2
Out[85]:
   0
A  3.0
```

(continues on next page)



(continued from previous page)

```
B 4.0
C NaN
D NaN
```

## Comparison operators

Comparison operators raise `ValueError` when `.index` are different.

**Previous Behavior** (Series):

Series compared values ignoring the `.index` as long as both had the same length:

```
In [1]: s1 == s2
Out[1]:
A    False
B     True
C    False
dtype: bool
```

**New behavior** (Series):

```
In [2]: s1 == s2
Out[2]:
ValueError: Can only compare identically-labeled Series objects
```

**Note:** To achieve the same result as previous versions (compare values based on locations ignoring `.index`), compare both `.values`.

```
In [86]: s1.values == s2.values
Out[86]: array([False,  True, False], dtype=bool)
```

If you want to compare Series aligning its `.index`, see flexible comparison methods section below:

```
In [87]: s1.eq(s2)
Out[87]:
A    False
B     True
C    False
D    False
dtype: bool
```

**Current Behavior** (DataFrame, no change):

```
In [3]: df1 == df2
Out[3]:
ValueError: Can only compare identically-labeled DataFrame objects
```

## Logical operators

Logical operators align both `.index` of left and right hand side.

**Previous behavior** (Series), only left hand side index was kept:

```
In [4]: s1 = pd.Series([True, False, True], index=list('ABC'))
In [5]: s2 = pd.Series([True, True, True], index=list('ABD'))
In [6]: s1 & s2
Out[6]:
A      True
B     False
C     False
dtype: bool
```

**New behavior** (Series):

```
In [88]: s1 = pd.Series([True, False, True], index=list('ABC'))
In [89]: s2 = pd.Series([True, True, True], index=list('ABD'))
In [90]: s1 & s2
Out[90]:
A      True
B     False
C     False
D     False
dtype: bool
```

---

**Note:** Series logical operators fill a NaN result with False.

---

---

**Note:** To achieve the same result as previous versions (compare values based on only left hand side index), you can use `reindex_like`:

```
In [91]: s1 & s2.reindex_like(s1)
Out[91]:
A      True
B     False
C     False
dtype: bool
```

---

**Current Behavior** (DataFrame, no change):

```
In [92]: df1 = pd.DataFrame([True, False, True], index=list('ABC'))
In [93]: df2 = pd.DataFrame([True, True, True], index=list('ABD'))
In [94]: df1 & df2
Out[94]:
0
A      True
B     False
C      NaN
D      NaN
```

## Flexible comparison methods

Series flexible comparison methods like `eq`, `ne`, `le`, `lt`, `ge` and `gt` now align both index. Use these operators if you want to compare two Series which has the different index.

```
In [95]: s1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])
In [96]: s2 = pd.Series([2, 2, 2], index=['b', 'c', 'd'])

In [97]: s1.eq(s2)
Out[97]:
a    False
b     True
c    False
d    False
dtype: bool

In [98]: s1.ge(s2)
Out[98]:
a    False
b     True
c     True
d    False
dtype: bool
```

Previously, this worked the same as comparison operators (see above).

### 1.12.2.3 Series type promotion on assignment

A Series will now correctly promote its dtype for assignment with incompat values to the current dtype ([GH13234](#))

```
In [99]: s = pd.Series()
```

**Previous behavior:**

```
In [2]: s["a"] = pd.Timestamp("2016-01-01")

In [3]: s["b"] = 3.0
TypeError: invalid type promotion
```

**New behavior:**

```
In [100]: s["a"] = pd.Timestamp("2016-01-01")
In [101]: s["b"] = 3.0

In [102]: s
Out[102]:
a    2016-01-01 00:00:00
b                      3
dtype: object

In [103]: s.dtype
Out[103]:
dtype('O')
```

#### 1.12.2.4 .to\_datetime() changes

Previously if `.to_datetime()` encountered mixed integers/floats and strings, but no datetimes with `errors='coerce'` it would convert all to `NaT`.

**Previous behavior:**

```
In [2]: pd.to_datetime([1, 'foo'], errors='coerce')
Out[2]: DatetimeIndex(['NaT', 'NaT'], dtype='datetime64[ns]', freq=None)
```

**Current behavior:**

This will now convert integers/floats with the default unit of `ns`.

```
In [104]: pd.to_datetime([1, 'foo'], errors='coerce')
Out[104]: DatetimeIndex(['1970-01-01 00:00:00.000000001', 'NaT'], dtype=
↳ 'datetime64[ns]', freq=None)
```

Bug fixes related to `.to_datetime()`:

- Bug in `pd.to_datetime()` when passing integers or floats, and no unit and `errors='coerce'` ([GH13180](#)).
- Bug in `pd.to_datetime()` when passing invalid datatypes (e.g. `bool`); will now respect the `errors` keyword ([GH13176](#))
- Bug in `pd.to_datetime()` which overflowed on `int8`, and `int16` dtypes ([GH13451](#))
- Bug in `pd.to_datetime()` raise `AttributeError` with `NaN` and the other string is not valid when `errors='ignore'` ([GH12424](#))
- Bug in `pd.to_datetime()` did not cast floats correctly when unit was specified, resulting in truncated datetime ([GH13834](#))

#### 1.12.2.5 Merging changes

Merging will now preserve the dtype of the join keys ([GH8596](#))

```
In [105]: df1 = pd.DataFrame({'key': [1], 'v1': [10]})

In [106]: df1
Out[106]:
   key  v1
0    1  10

In [107]: df2 = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})

In [108]: df2
Out[108]:
   key  v1
0    1  20
1    2  30
```

**Previous behavior:**

```
In [5]: pd.merge(df1, df2, how='outer')
Out[5]:
   key    v1
0  1.0  10.0
```

(continues on next page)

(continued from previous page)

```

1  1.0  20.0
2  2.0  30.0

In [6]: pd.merge(df1, df2, how='outer').dtypes
Out[6]:
key      float64
v1       float64
dtype: object

```

**New behavior:**

We are able to preserve the join keys

```

In [109]: pd.merge(df1, df2, how='outer')
Out[109]:
   key  v1
0    1  10
1    1  20
2    2  30

In [110]: pd.merge(df1, df2, how='outer').dtypes
Out[110]:
key      int64
v1       int64
dtype: object

```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast, which is unchanged from previous.

```

In [111]: pd.merge(df1, df2, how='outer', on='key')
Out[111]:
   key  v1_x  v1_y
0    1  10.0   20
1    2   NaN   30

In [112]: pd.merge(df1, df2, how='outer', on='key').dtypes
Out[112]:
key      int64
v1_x     float64
v1_y     int64
dtype: object

```

**1.12.2.6 .describe() changes**

Percentile identifiers in the index of a `.describe()` output will now be rounded to the least precision that keeps them distinct ([GH13104](#))

```

In [113]: s = pd.Series([0, 1, 2, 3, 4])

In [114]: df = pd.DataFrame([0, 1, 2, 3, 4])

```

**Previous behavior:**

The percentiles were rounded to at most one decimal place, which could raise `ValueError` for a data frame if the percentiles were duplicated.

```
In [3]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
```

```
Out [3]:
```

```
count      5.000000
mean       2.000000
std        1.581139
min        0.000000
0.0%       0.000400
0.1%       0.002000
0.1%       0.004000
50%        2.000000
99.9%      3.996000
100.0%     3.998000
100.0%     3.999600
max        4.000000
dtype: float64
```

```
In [4]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
```

```
Out [4]:
```

```
...
ValueError: cannot reindex from a duplicate axis
```

#### New behavior:

```
In [115]: s.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
```

```
Out [115]:
```

```
count      5.000000
mean       2.000000
std        1.581139
min        0.000000
0.01%      0.000400
0.05%      0.002000
0.1%       0.004000
50%        2.000000
99.9%      3.996000
99.95%     3.998000
99.99%     3.999600
max        4.000000
dtype: float64
```

```
In [116]: df.describe(percentiles=[0.0001, 0.0005, 0.001, 0.999, 0.9995, 0.9999])
```

```
////////////////////////////////////
```

```
↪
      0
count  5.000000
mean   2.000000
std    1.581139
min    0.000000
0.01%  0.000400
0.05%  0.002000
0.1%   0.004000
50%    2.000000
99.9%  3.996000
99.95% 3.998000
99.99% 3.999600
max     4.000000
```

Furthermore:

- Passing duplicated percentiles will now raise a `ValueError`.
- Bug in `.describe()` on a `DataFrame` with a mixed-dtype column index, which would previously raise a `TypeError` ([GH13288](#))

### 1.12.2.7 Period changes

#### PeriodIndex now has period dtype

`PeriodIndex` now has its own period dtype. The period dtype is a pandas extension dtype like category or the *timezone aware dtype* (`datetime64[ns, tz]`) ([GH13941](#)). As a consequence of this change, `PeriodIndex` no longer has an integer dtype:

##### Previous behavior:

```
In [1]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')

In [2]: pi
Out[2]: PeriodIndex(['2016-08-01'], dtype='int64', freq='D')

In [3]: pd.api.types.is_integer_dtype(pi)
Out[3]: True

In [4]: pi.dtype
Out[4]: dtype('int64')
```

##### New behavior:

```
In [117]: pi = pd.PeriodIndex(['2016-08-01'], freq='D')

In [118]: pi
Out[118]: PeriodIndex(['2016-08-01'], dtype='period[D]', freq='D')

In [119]: pd.api.types.is_integer_dtype(pi)
Out[119]: False

In [120]: pd.api.types.is_period_dtype(pi)
Out[120]: True

In [121]: pi.dtype
Out[121]: period[D]

In [122]: type(pi.dtype)
Out[122]: pandas.core.dtypes.dtypes.PeriodDtype
```

#### Period('NaT') now returns pd.NaT

Previously, `Period` has its own `Period('NaT')` representation different from `pd.NaT`. Now `Period('NaT')` has been changed to return `pd.NaT`. ([GH12759](#), [GH13582](#))

##### Previous behavior:

```
In [5]: pd.Period('NaT', freq='D')
Out[5]: Period('NaT', 'D')
```

**New behavior:**

These result in `pd.NaT` without providing `freq` option.

```
In [123]: pd.Period('NaT')
Out[123]: NaT

In [124]: pd.Period(None)
Out[124]: NaT
```

To be compatible with `Period` addition and subtraction, `pd.NaT` now supports addition and subtraction with `int`. Previously it raised `ValueError`.

**Previous behavior:**

```
In [5]: pd.NaT + 1
...
ValueError: Cannot add integral value to Timestamp without freq.
```

**New behavior:**

```
In [125]: pd.NaT + 1
Out[125]: NaT

In [126]: pd.NaT - 1
Out[126]: NaT
```

### **`PeriodIndex.values` now returns array of `Period` object**

`.values` is changed to return an array of `Period` objects, rather than an array of integers ([GH13988](#)).

**Previous behavior:**

```
In [6]: pi = pd.PeriodIndex(['2011-01', '2011-02'], freq='M')
In [7]: pi.values
array([492, 493])
```

**New behavior:**

```
In [127]: pi = pd.PeriodIndex(['2011-01', '2011-02'], freq='M')
In [128]: pi.values
Out[128]: array([Period('2011-01', 'M'), Period('2011-02', 'M')], dtype=object)
```

### **1.12.2.8 Index + / – no longer used for set operations**

Addition and subtraction of the base `Index` type and of `DatetimeIndex` (not the numeric index types) previously performed set operations (set union and difference). This behavior was already deprecated since 0.15.0 (in favor using the specific `.union()` and `.difference()` methods), and is now disabled. When possible, `+` and `-` are now used for element-wise operations, for example for concatenating strings or subtracting datetimes ([GH8227](#), [GH14127](#)).

Previous behavior:



```
In [1]: pd.Index(['a', 'b']) + pd.Index(['a', 'c'])
FutureWarning: using '+' to provide set union with Indexes is deprecated, use '|' or .
↳union()
Out[1]: Index(['a', 'b', 'c'], dtype='object')
```

**New behavior:** the same operation will now perform element-wise addition:

```
In [129]: pd.Index(['a', 'b']) + pd.Index(['a', 'c'])
Out[129]: Index(['aa', 'bc'], dtype='object')
```

Note that numeric Index objects already performed element-wise operations. For example, the behavior of adding two integer Indexes is unchanged. The base Index is now made consistent with this behavior.

```
In [130]: pd.Index([1, 2, 3]) + pd.Index([2, 3, 4])
Out[130]: Int64Index([3, 5, 7], dtype='int64')
```

Further, because of this change, it is now possible to subtract two DatetimeIndex objects resulting in a TimedeltaIndex:

**Previous behavior:**

```
In [1]: pd.DatetimeIndex(['2016-01-01', '2016-01-02']) - pd.DatetimeIndex(['2016-01-02', '2016-01-03'])
↳FutureWarning: using '-' to provide set differences with datetimelike Indexes is
↳deprecated, use .difference()
Out[1]: DatetimeIndex(['2016-01-01'], dtype='datetime64[ns]', freq=None)
```

**New behavior:**

```
In [131]: pd.DatetimeIndex(['2016-01-01', '2016-01-02']) - pd.DatetimeIndex(['2016-01-02', '2016-01-03'])
↳Out[131]: TimedeltaIndex(['-1 days', '-1 days'], dtype='timedelta64[ns]', freq=None)
```

### 1.12.2.9 Index.difference and .symmetric\_difference changes

Index.difference and Index.symmetric\_difference will now, more consistently, treat NaN values as any other values. (GH13514)

```
In [132]: idx1 = pd.Index([1, 2, 3, np.nan])
In [133]: idx2 = pd.Index([0, 1, np.nan])
```

**Previous behavior:**

```
In [3]: idx1.difference(idx2)
Out[3]: Float64Index([nan, 2.0, 3.0], dtype='float64')

In [4]: idx1.symmetric_difference(idx2)
Out[4]: Float64Index([0.0, nan, 2.0, 3.0], dtype='float64')
```

**New behavior:**

```
In [134]: idx1.difference(idx2)
Out[134]: Float64Index([2.0, 3.0], dtype='float64')

In [135]: idx1.symmetric_difference(idx2)
Out[135]: Float64Index([0.0, 2.0, 3.0], dtype='float64')
```

(continues on next page)

### 1.12.2.10 `Index.unique` consistently returns `Index`

`Index.unique()` now returns unique values as an `Index` of the appropriate dtype. (GH13395). Previously, most `Index` classes returned `np.ndarray`, and `DatetimeIndex`, `TimedeltaIndex` and `PeriodIndex` returned `Index` to keep metadata like timezone.

#### Previous behavior:

```
In [1]: pd.Index([1, 2, 3]).unique()
Out[1]: array([1, 2, 3])

In [2]: pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], tz='Asia/Tokyo').
↳unique()
Out[2]:
DatetimeIndex(['2011-01-01 00:00:00+09:00', '2011-01-02 00:00:00+09:00',
                '2011-01-03 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq=None)
```

#### New behavior:

```
In [136]: pd.Index([1, 2, 3]).unique()
Out[136]: Int64Index([1, 2, 3], dtype='int64')

In [137]: pd.DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], tz='Asia/Tokyo
↳').unique()
Out[137]:
DatetimeIndex(['2011-01-01 00:00:00+09:00', '2011-01-02 00:00:00+09:00',
                '2011-01-03 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq=None)
```

### 1.12.2.11 `MultiIndex` constructors, `groupby` and `set_index` preserve categorical dtypes

`MultiIndex.from_arrays` and `MultiIndex.from_product` will now preserve categorical dtype in `MultiIndex` levels (GH13743, GH13854).

```
In [138]: cat = pd.Categorical(['a', 'b'], categories=list("bac"))

In [139]: lvl1 = ['foo', 'bar']

In [140]: midx = pd.MultiIndex.from_arrays([cat, lvl1])

In [141]: midx
Out[141]:
MultiIndex(levels=[['b', 'a', 'c'], ['bar', 'foo']],
            labels=[[1, 0], [1, 0]])
```

#### Previous behavior:

```
In [4]: midx.levels[0]
Out[4]: Index(['b', 'a', 'c'], dtype='object')

In [5]: midx.get_level_values[0]
Out[5]: Index(['a', 'b'], dtype='object')
```

**New behavior:** the single level is now a CategoricalIndex:

```
In [142]: midx.levels[0]
Out[142]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
↳ dtype='category')

In [143]: midx.get_level_values(0)
////////////////////////////////////
↳ CategoricalIndex(['a', 'b'], categories=['b', 'a', 'c'], ordered=False, dtype=
↳ 'category')
```

An analogous change has been made to `MultiIndex.from_product`. As a consequence, `groupby` and `set_index` also preserve categorical dtypes in indexes

```
In [144]: df = pd.DataFrame({'A': [0, 1], 'B': [10, 11], 'C': cat})

In [145]: df_grouped = df.groupby(by=['A', 'C']).first()

In [146]: df_set_idx = df.set_index(['A', 'C'])
```

**Previous behavior:**

```
In [11]: df_grouped.index.levels[1]
Out[11]: Index(['b', 'a', 'c'], dtype='object', name='C')
In [12]: df_grouped.reset_index().dtypes
Out[12]:
A      int64
C      object
B      float64
dtype: object

In [13]: df_set_idx.index.levels[1]
Out[13]: Index(['b', 'a', 'c'], dtype='object', name='C')
In [14]: df_set_idx.reset_index().dtypes
Out[14]:
A      int64
C      object
B      int64
dtype: object
```

**New behavior:**

```
In [147]: df_grouped.index.levels[1]
Out[147]: CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False,
↳ name='C', dtype='category')

In [148]: df_grouped.reset_index().dtypes
////////////////////////////////////
↳
A      int64
C      category
B      float64
dtype: object

In [149]: df_set_idx.index.levels[1]
////////////////////////////////////
↳ CategoricalIndex(['b', 'a', 'c'], categories=['b', 'a', 'c'], ordered=False, name='C
↳ ', dtype='category')
```

(continues on next page)

(continued from previous page)

```
In [150]: df_set_idx.reset_index().dtypes
```

```

////////////////////////////////////
↪
A      int64
C      category
B      int64
dtype: object

```

### 1.12.2.12 read\_csv will progressively enumerate chunks

When `read_csv()` is called with `chunksize=n` and without specifying an index, each chunk used to have an independently generated index from 0 to `n-1`. They are now given instead a progressive index, starting from 0 for the first chunk, from `n` for the second, and so on, so that, when concatenated, they are identical to the result of calling `read_csv()` without the `chunksize=` argument (GH12185).

```
In [151]: data = 'A,B\n0,1\n2,3\n4,5\n6,7'
```

#### Previous behavior:

```
In [2]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
```

```
Out[2]:
```

```

   A  B
0  0  1
1  2  3
0  4  5
1  6  7

```

#### New behavior:

```
In [152]: pd.concat(pd.read_csv(StringIO(data), chunksize=2))
```

```
Out[152]:
```

```

   A  B
0  0  1
1  2  3
2  4  5
3  6  7

```

### 1.12.2.13 Sparse Changes

These changes allow pandas to handle sparse data with more dtypes, and for work to make a smoother experience with data handling.

#### int64 and bool support enhancements

Sparse data structures now gained enhanced support of `int64` and `bool` dtype (GH667, GH13849).

Previously, sparse data were `float64` dtype by default, even if all inputs were of `int` or `bool` dtype. You had to specify dtype explicitly to create sparse data with `int64` dtype. Also, `fill_value` had to be specified explicitly because the default was `np.nan` which doesn't appear in `int64` or `bool` data.

```

In [1]: pd.SparseArray([1, 2, 0, 0])
Out[1]:
[1.0, 2.0, 0.0, 0.0]
Fill: nan
IntIndex
Indices: array([0, 1, 2, 3], dtype=int32)

# specifying int64 dtype, but all values are stored in sp_values because
# fill_value default is np.nan
In [2]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64)
Out[2]:
[1, 2, 0, 0]
Fill: nan
IntIndex
Indices: array([0, 1, 2, 3], dtype=int32)

In [3]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64, fill_value=0)
Out[3]:
[1, 2, 0, 0]
Fill: 0
IntIndex
Indices: array([0, 1], dtype=int32)

```

As of v0.19.0, sparse data keeps the input dtype, and uses more appropriate `fill_value` defaults (0 for `int64` dtype, `False` for `bool` dtype).

```

In [153]: pd.SparseArray([1, 2, 0, 0], dtype=np.int64)
Out[153]:
[1, 2, 0, 0]
Fill: 0
IntIndex
Indices: array([0, 1], dtype=int32)

In [154]: pd.SparseArray([True, False, False, False])
\\Out[154]:
↳
[True, False, False, False]
Fill: False
IntIndex
Indices: array([0], dtype=int32)

```

See the [docs](#) for more details.

## Operators now preserve dtypes

- Sparse data structure now can preserve dtype after arithmetic ops ([GH13848](#))

```

In [155]: s = pd.SparseSeries([0, 2, 0, 1], fill_value=0, dtype=np.int64)

In [156]: s.dtype
Out[156]: dtype('int64')

In [157]: s + 1
\\Out[157]:
0    1
1    3

```

(continues on next page)

(continued from previous page)

```

2      1
3      2
dtype: int64
BlockIndex
Block locations: array([1, 3], dtype=int32)
Block lengths: array([1, 1], dtype=int32)

```

- Sparse data structure now support `astype` to convert internal dtype ([GH13900](#))

```
In [158]: s = pd.SparseSeries([1., 0., 2., 0.], fill_value=0)
```

```
In [159]: s
```

```
Out[159]:
```

```

0      1.0
1      0.0
2      2.0
3      0.0
dtype: float64
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 1], dtype=int32)

```

```
In [160]: s.astype(np.int64)
```

```

////////////////////////////////////
↪
0      1
1      0
2      2
3      0
dtype: int64
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 1], dtype=int32)

```

`astype` fails if data contains values which cannot be converted to specified dtype. Note that the limitation is applied to `fill_value` which default is `np.nan`.

```
In [7]: pd.SparseSeries([1., np.nan, 2., np.nan], fill_value=np.nan).astype(np.
↪int64)
```

```
Out [7]:
```

```
ValueError: unable to coerce current fill_value nan to int64 dtype
```

## Other sparse fixes

- Subclassed `SparseDataFrame` and `SparseSeries` now preserve class types when slicing or transposing. ([GH13787](#))
- `SparseArray` with `bool` dtype now supports logical (`bool`) operators ([GH14000](#))
- Bug in `SparseSeries` with `MultiIndex []` indexing may raise `IndexError` ([GH13144](#))
- Bug in `SparseSeries` with `MultiIndex []` indexing result may have normal `Index` ([GH13144](#))
- Bug in `SparseDataFrame` in which `axis=None` did not default to `axis=0` ([GH13048](#))
- Bug in `SparseSeries` and `SparseDataFrame` creation with `object` dtype may raise `TypeError` ([GH11633](#))

- Bug in `SparseDataFrame` doesn't respect passed `SparseArray` or `SparseSeries`'s `dtype` and `fill_value` (GH13866)
- Bug in `SparseArray` and `SparseSeries` don't apply `ufunc` to `fill_value` (GH13853)
- Bug in `SparseSeries.abs` incorrectly keeps negative `fill_value` (GH13853)
- Bug in single row slicing on multi-type `SparseDataFrame`s, types were previously forced to float (GH13917)
- Bug in `SparseSeries` slicing changes integer `dtype` to float (GH8292)
- Bug in `SparseDataFrame` comparison ops may raise `TypeError` (GH13001)
- Bug in `SparseDataFrame.isnull` raises `ValueError` (GH8276)
- Bug in `SparseSeries` representation with `bool` `dtype` may raise `IndexError` (GH13110)
- Bug in `SparseSeries` and `SparseDataFrame` of `bool` or `int64` `dtype` may display its values like `float64` `dtype` (GH13110)
- Bug in sparse indexing using `SparseArray` with `bool` `dtype` may return incorrect result (GH13985)
- Bug in `SparseArray` created from `SparseSeries` may lose `dtype` (GH13999)
- Bug in `SparseSeries` comparison with dense returns normal `Series` rather than `SparseSeries` (GH13999)

#### 1.12.2.14 Indexer dtype changes

**Note:** This change only affects 64 bit python running on Windows, and only affects relatively advanced indexing operations

Methods such as `Index.get_indexer` that return an indexer array, coerce that array to a “platform int”, so that it can be directly used in 3rd party library operations like `numpy.take`. Previously, a platform int was defined as `np.int_` which corresponds to a C integer, but the correct type, and what is being used now, is `np.intp`, which corresponds to the C integer size that can hold a pointer (GH3033, GH13972).

These types are the same on many platform, but for 64 bit python on Windows, `np.int_` is 32 bits, and `np.intp` is 64 bits. Changing this behavior improves performance for many operations on that platform.

**Previous behavior:**

```
In [1]: i = pd.Index(['a', 'b', 'c'])
In [2]: i.get_indexer(['b', 'b', 'c']).dtype
Out[2]: dtype('int32')
```

**New behavior:**

```
In [1]: i = pd.Index(['a', 'b', 'c'])
In [2]: i.get_indexer(['b', 'b', 'c']).dtype
Out[2]: dtype('int64')
```

### 1.12.2.15 Other API Changes

- `Timestamp.to_pydatetime` will issue a `UserWarning` when `warn=True`, and the instance has a non-zero number of nanoseconds, previously this would print a message to `stdout` ([GH14101](#)).
- `Series.unique()` with `datetime` and `timezone` now returns `array of Timestamp with timezone` ([GH13565](#)).
- `Panel.to_sparse()` will raise a `NotImplementedError` exception when called ([GH13778](#)).
- `Index.reshape()` will raise a `NotImplementedError` exception when called ([GH12882](#)).
- `.filter()` enforces mutual exclusion of the keyword arguments ([GH12399](#)).
- `eval`'s upcasting rules for `float32` types have been updated to be more consistent with NumPy's rules. New behavior will not upcast to `float64` if you multiply a pandas `float32` object by a scalar `float64` ([GH12388](#)).
- An `UnsupportedFunctionCall` error is now raised if NumPy ufuncs like `np.mean` are called on `groupby` or `resample` objects ([GH12811](#)).
- `__setitem__` will no longer apply a callable `rhs` as a function instead of storing it. Call `where` directly to get the previous behavior ([GH13299](#)).
- Calls to `.sample()` will respect the random seed set via `numpy.random.seed(n)` ([GH13161](#)).
- `Styler.apply` is now more strict about the outputs your function must return. For `axis=0` or `axis=1`, the output shape must be identical. For `axis=None`, the output must be a `DataFrame` with identical columns and index labels ([GH13222](#)).
- `Float64Index.astype(int)` will now raise `ValueError` if `Float64Index` contains `NaN` values ([GH13149](#)).
- `TimedeltaIndex.astype(int)` and `DatetimeIndex.astype(int)` will now return `Int64Index` instead of `np.array` ([GH13209](#)).
- Passing `Period` with multiple frequencies to normal `Index` now returns `Index` with `object` dtype ([GH13664](#)).
- `PeriodIndex.fillna` with `Period` has different `freq` now coerces to `object` dtype ([GH13664](#)).
- Faceted boxplots from `DataFrame.boxplot(by=col)` now return a `Series` when `return_type` is not `None`. Previously these returned an `OrderedDict`. Note that when `return_type=None`, the default, these still return a 2-D NumPy array ([GH12216](#), [GH7096](#)).
- `pd.read_hdf` will now raise a `ValueError` instead of `KeyError`, if a mode other than `r`, `r+` and `a` is supplied. ([GH13623](#))
- `pd.read_csv()`, `pd.read_table()`, and `pd.read_hdf()` raise the builtin `FileNotFoundError` exception for Python 3.x when called on a nonexistent file; this is back-ported as `IOError` in Python 2.x ([GH14086](#))
- More informative exceptions are passed through the csv parser. The exception type would now be the original exception type instead of `CParserError` ([GH13652](#)).
- `pd.read_csv()` in the C engine will now issue a `ParserWarning` or raise a `ValueError` when `sep` encoded is more than one character long ([GH14065](#))
- `DataFrame.values` will now return `float64` with a `DataFrame` of mixed `int64` and `uint64` dtypes, conforming to `np.find_common_type` ([GH10364](#), [GH13917](#))
- `.groupby.groups` will now return a dictionary of `Index` objects, rather than a dictionary of `np.ndarray` or `lists` ([GH14293](#))



### 1.12.3 Deprecations

- `Series.reshape` and `Categorical.reshape` have been deprecated and will be removed in a subsequent release ([GH12882](#), [GH12882](#))
- `PeriodIndex.to_datetime` has been deprecated in favor of `PeriodIndex.to_timestamp` ([GH8254](#))
- `Timestamp.to_datetime` has been deprecated in favor of `Timestamp.to_pydatetime` ([GH8254](#))
- `Index.to_datetime` and `DatetimeIndex.to_datetime` have been deprecated in favor of `pd.to_datetime` ([GH8254](#))
- `pandas.core.datetools` module has been deprecated and will be removed in a subsequent release ([GH14094](#))
- `SparseList` has been deprecated and will be removed in a future version ([GH13784](#))
- `DataFrame.to_html()` and `DataFrame.to_latex()` have dropped the `colSpace` parameter in favor of `col_space` ([GH13857](#))
- `DataFrame.to_sql()` has deprecated the `flavor` parameter, as it is superfluous when SQLAlchemy is not installed ([GH13611](#))
- Depreciated `read_csv` keywords:
  - `compact_ints` and `use_unsigned` have been deprecated and will be removed in a future version ([GH13320](#))
  - `buffer_lines` has been deprecated and will be removed in a future version ([GH13360](#))
  - `as_reccarray` has been deprecated and will be removed in a future version ([GH13373](#))
  - `skip_footer` has been deprecated in favor of `skipfooter` and will be removed in a future version ([GH13349](#))
- top-level `pd.ordered_merge()` has been renamed to `pd.merge_ordered()` and the original name will be removed in a future version ([GH13358](#))
- `Timestamp.offset` property (and named arg in the constructor), has been deprecated in favor of `freq` ([GH12160](#))
- `pd.tseries.util.pivot_annual` is deprecated. Use `pivot_table` as alternative, an example is [here](#) ([GH736](#))
- `pd.tseries.util.isleapyear` has been deprecated and will be removed in a subsequent release. `Datetime`-likes now have a `.is_leap_year` property ([GH13727](#))
- `Panel4D` and `PanelND` constructors are deprecated and will be removed in a future version. The recommended way to represent these types of n-dimensional data are with the [xarray package](#). Pandas provides a `to_xarray()` method to automate this conversion ([GH13564](#)).
- `pandas.tseries.frequencies.get_standard_freq` is deprecated. Use `pandas.tseries.frequencies.to_offset(freq).rule_code` instead ([GH13874](#))
- `pandas.tseries.frequencies.to_offset`'s `freqstr` keyword is deprecated in favor of `freq` ([GH13874](#))
- `Categorical.from_array` has been deprecated and will be removed in a future version ([GH13854](#))

### 1.12.4 Removal of prior version deprecations/changes

- The `SparsePanel` class has been removed ([GH13778](#))

- The `pd.sandbox` module has been removed in favor of the external library `pandas-qt` ([GH13670](#))
- The `pandas.io.data` and `pandas.io.wb` modules are removed in favor of the `pandas-datareader` package ([GH13724](#)).
- The `pandas.tools.rplot` module has been removed in favor of the `seaborn` package ([GH13855](#))
- `DataFrame.to_csv()` has dropped the `engine` parameter, as was deprecated in 0.17.1 ([GH11274](#), [GH13419](#))
- `DataFrame.to_dict()` has dropped the `outtype` parameter in favor of `orient` ([GH13627](#), [GH8486](#))
- `pd.Categorical` has dropped setting of the `ordered` attribute directly in favor of the `set_ordered` method ([GH13671](#))
- `pd.Categorical` has dropped the `levels` attribute in favor of `categories` ([GH8376](#))
- `DataFrame.to_sql()` has dropped the `mysql` option for the `flavor` parameter ([GH13611](#))
- `Panel.shift()` has dropped the `lags` parameter in favor of `periods` ([GH14041](#))
- `pd.Index` has dropped the `diff` method in favor of `difference` ([GH13669](#))
- `pd.DataFrame` has dropped the `to_wide` method in favor of `to_panel` ([GH14039](#))
- `Series.to_csv` has dropped the `nanRep` parameter in favor of `na_rep` ([GH13804](#))
- `Series.xs`, `DataFrame.xs`, `Panel.xs`, `Panel.major_xs`, and `Panel.minor_xs` have dropped the `copy` parameter ([GH13781](#))
- `str.split` has dropped the `return_type` parameter in favor of `expand` ([GH13701](#))
- Removal of the legacy time rules (offset aliases), deprecated since 0.17.0 (this has been alias since 0.8.0) ([GH13590](#), [GH13868](#)). Now legacy time rules raises `ValueError`. For the list of currently supported off-sets, see [here](#).
- The default value for the `return_type` parameter for `DataFrame.plot.box` and `DataFrame.boxplot` changed from `None` to `"axes"`. These methods will now return a matplotlib axes by default instead of a dictionary of artists. See [here](#) ([GH6581](#)).
- The `tquery` and `uquery` functions in the `pandas.io.sql` module are removed ([GH5950](#)).

### 1.12.5 Performance Improvements

- Improved performance of sparse `IntIndex.intersect` ([GH13082](#))
- Improved performance of sparse arithmetic with `BlockIndex` when the number of blocks are large, though recommended to use `IntIndex` in such cases ([GH13082](#))
- Improved performance of `DataFrame.quantile()` as it now operates per-block ([GH11623](#))
- Improved performance of float64 hash table operations, fixing some very slow indexing and groupby operations in python 3 ([GH13166](#), [GH13334](#))
- Improved performance of `DataFrameGroupBy.transform` ([GH12737](#))
- Improved performance of `Index` and `Series.duplicated` ([GH10235](#))
- Improved performance of `Index.difference` ([GH12044](#))
- Improved performance of `RangeIndex.is_monotonic_increasing` and `is_monotonic_decreasing` ([GH13749](#))
- Improved performance of datetime string parsing in `DatetimeIndex` ([GH13692](#))
- Improved performance of hashing `Period` ([GH12817](#))

- Improved performance of `factorize` of datetime with timezone (GH13750)
- Improved performance of by lazily creating indexing hashtables on larger Indexes (GH14266)
- Improved performance of `groupby.groups` (GH14293)
- Unnecessary materializing of a MultiIndex when introspecting for memory usage (GH14308)

### 1.12.6 Bug Fixes

- Bug in `groupby().shift()`, which could cause a segfault or corruption in rare circumstances when grouping by columns with missing values (GH13813)
- Bug in `groupby().cumsum()` calculating `cumprod` when `axis=1`. (GH13994)
- Bug in `pd.to_timedelta()` in which the `errors` parameter was not being respected (GH13613)
- Bug in `io.json.json_normalize()`, where non-ascii keys raised an exception (GH13213)
- Bug when passing a not-default-indexed Series as `xerr` or `yerr` in `.plot()` (GH11858)
- Bug in area plot draws legend incorrectly if subplot is enabled or legend is moved after plot (matplotlib 1.5.0 is required to draw area plot legend properly) (GH9161, GH13544)
- Bug in DataFrame assignment with an object-dtyped Index where the resultant column is mutable to the original object. (GH13522)
- Bug in matplotlib `AutoDataFormatter`; this restores the second scaled formatting and re-adds micro-second scaled formatting (GH13131)
- Bug in selection from a `HDFStore` with a fixed format and `start` and/or `stop` specified will now return the selected range (GH8287)
- Bug in `Categorical.from_codes()` where an unhelpful error was raised when an invalid ordered parameter was passed in (GH14058)
- Bug in Series construction from a tuple of integers on windows not returning default dtype (`int64`) (GH13646)
- Bug in `TimedeltaIndex` addition with a Datetime-like object where addition overflow was not being caught (GH14068)
- Bug in `.groupby(...).resample(...)` when the same object is called multiple times (GH13174)
- Bug in `.to_records()` when index name is a unicode string (GH13172)
- Bug in calling `.memory_usage()` on object which doesn't implement (GH12924)
- Regression in `Series.quantile` with nans (also shows up in `.median()` and `.describe()`); furthermore now names the Series with the quantile (GH13098, GH13146)
- Bug in `SeriesGroupBy.transform` with datetime values and missing groups (GH13191)
- Bug where empty Series were incorrectly coerced in datetime-like numeric operations (GH13844)
- Bug in `Categorical` constructor when passed a `Categorical` containing datetimes with timezones (GH14190)
- Bug in `Series.str.extractall()` with `str` index raises `ValueError` (GH13156)
- Bug in `Series.str.extractall()` with single group and quantifier (GH13382)
- Bug in `DatetimeIndex` and `Period` subtraction raises `ValueError` or `AttributeError` rather than `TypeError` (GH13078)
- Bug in Index and Series created with NaN and NaT mixed data may not have `datetime64` dtype (GH13324)

- Bug in Index and Series may ignore `np.datetime64('nat')` and `np.timedelta64('nat')` to infer dtype ([GH13324](#))
- Bug in PeriodIndex and Period subtraction raises `AttributeError` ([GH13071](#))
- Bug in PeriodIndex construction returning a float64 index in some circumstances ([GH13067](#))
- Bug in `.resample(...)` with a PeriodIndex not changing its freq appropriately when empty ([GH13067](#))
- Bug in `.resample(...)` with a PeriodIndex not retaining its type or name with an empty DataFrame appropriately when empty ([GH13212](#))
- Bug in `groupby(...).apply(...)` when the passed function returns scalar values per group ([GH13468](#)).
- Bug in `groupby(...).resample(...)` where passing some keywords would raise an exception ([GH13235](#))
- Bug in `.tz_convert` on a tz-aware DateTimeIndex that relied on index being sorted for correct results ([GH13306](#))
- Bug in `.tz_localize` with `dateutil.tz.tzlocal` may return incorrect result ([GH13583](#))
- Bug in `DatetimeTZDtype` dtype with `dateutil.tz.tzlocal` cannot be regarded as valid dtype ([GH13583](#))
- Bug in `pd.read_hdf()` where attempting to load an HDF file with a single dataset, that had one or more categorical columns, failed unless the key argument was set to the name of the dataset. ([GH13231](#))
- Bug in `.rolling()` that allowed a negative integer window in construction of the `Rolling()` object, but would later fail on aggregation ([GH13383](#))
- Bug in Series indexing with tuple-valued data and a numeric index ([GH13509](#))
- Bug in printing `pd.DataFrame` where unusual elements with the `object` dtype were causing segfaults ([GH13717](#))
- Bug in ranking Series which could result in segfaults ([GH13445](#))
- Bug in various index types, which did not propagate the name of passed index ([GH12309](#))
- Bug in `DatetimeIndex`, which did not honour the `copy=True` ([GH13205](#))
- Bug in `DatetimeIndex.is_normalized` returns incorrectly for normalized date\_range in case of local timezones ([GH13459](#))
- Bug in `pd.concat` and `.append` may coerces `datetime64` and `timedelta` to `object` dtype containing python built-in `datetime` or `timedelta` rather than `Timestamp` or `Timedelta` ([GH13626](#))
- Bug in `PeriodIndex.append` may raises `AttributeError` when the result is `object` dtype ([GH13221](#))
- Bug in `CategoricalIndex.append` may accept normal list ([GH13626](#))
- Bug in `pd.concat` and `.append` with the same timezone get reset to UTC ([GH7795](#))
- Bug in Series and DataFrame `.append` raises `AmbiguousTimeError` if data contains datetime near DST boundary ([GH13626](#))
- Bug in `DataFrame.to_csv()` in which float values were being quoted even though quotations were specified for non-numeric values only ([GH12922](#), [GH13259](#))
- Bug in `DataFrame.describe()` raising `ValueError` with only boolean columns ([GH13898](#))
- Bug in `MultiIndex` slicing where extra elements were returned when level is non-unique ([GH12896](#))
- Bug in `.str.replace` does not raise `TypeError` for invalid replacement ([GH13438](#))
- Bug in `MultiIndex.from_arrays` which didn't check for input array lengths matching ([GH13599](#))

- Bug in `cartesian_product` and `MultiIndex.from_product` which may raise with empty input arrays ([GH12258](#))
- Bug in `pd.read_csv()` which may cause a segfault or corruption when iterating in large chunks over a stream/file under rare circumstances ([GH13703](#))
- Bug in `pd.read_csv()` which caused errors to be raised when a dictionary containing scalars is passed in for `na_values` ([GH12224](#))
- Bug in `pd.read_csv()` which caused BOM files to be incorrectly parsed by not ignoring the BOM ([GH4793](#))
- Bug in `pd.read_csv()` with `engine='python'` which raised errors when a numpy array was passed in for `usecols` ([GH12546](#))
- Bug in `pd.read_csv()` where the index columns were being incorrectly parsed when parsed as dates with a `thousands` parameter ([GH14066](#))
- Bug in `pd.read_csv()` with `engine='python'` in which NaN values weren't being detected after data was converted to numeric values ([GH13314](#))
- Bug in `pd.read_csv()` in which the `nrows` argument was not properly validated for both engines ([GH10476](#))
- Bug in `pd.read_csv()` with `engine='python'` in which infinities of mixed-case forms were not being interpreted properly ([GH13274](#))
- Bug in `pd.read_csv()` with `engine='python'` in which trailing NaN values were not being parsed ([GH13320](#))
- Bug in `pd.read_csv()` with `engine='python'` when reading from a `tempfile.TemporaryFile` on Windows with Python 3 ([GH13398](#))
- Bug in `pd.read_csv()` that prevents `usecols` kwarg from accepting single-byte unicode strings ([GH13219](#))
- Bug in `pd.read_csv()` that prevents `usecols` from being an empty set ([GH13402](#))
- Bug in `pd.read_csv()` in the C engine where the NULL character was not being parsed as NULL ([GH14012](#))
- Bug in `pd.read_csv()` with `engine='c'` in which NULL `quotechar` was not accepted even though quoting was specified as `None` ([GH13411](#))
- Bug in `pd.read_csv()` with `engine='c'` in which fields were not properly cast to float when quoting was specified as non-numeric ([GH13411](#))
- Bug in `pd.read_csv()` in Python 2.x with non-UTF8 encoded, multi-character separated data ([GH3404](#))
- Bug in `pd.read_csv()`, where aliases for utf-xx (e.g. UTF-xx, UTF\_xx, utf\_xx) raised `UnicodeDecodeError` ([GH13549](#))
- Bug in `pd.read_csv`, `pd.read_table`, `pd.read_fwf`, `pd.read_stata` and `pd.read_sas` where files were opened by parsers but not closed if both `chunksize` and `iterator` were `None`. ([GH13940](#))
- Bug in `StataReader`, `StataWriter`, `XportReader` and `SAS7BDATReader` where a file was not properly closed when an error was raised. ([GH13940](#))
- Bug in `pd.pivot_table()` where `margins_name` is ignored when `aggfunc` is a list ([GH13354](#))
- Bug in `pd.Series.str.zfill`, `center`, `ljust`, `rjust`, and `pad` when passing non-integers, did not raise `TypeError` ([GH13598](#))
- Bug in checking for any null objects in a `TimedeltaIndex`, which always returned `True` ([GH13603](#))
- Bug in `Series` arithmetic raises `TypeError` if it contains datetime-like as object dtype ([GH13043](#))

- Bug `Series.isnull()` and `Series.notnull()` ignore `Period('NaT')` (GH13737)
- Bug `Series.fillna()` and `Series.dropna()` don't affect to `Period('NaT')` (GH13737)
- Bug in `.fillna(value=np.nan)` incorrectly raises `KeyError` on a category dtype Series (GH14021)
- Bug in extension dtype creation where the created types were not is/identical (GH13285)
- Bug in `.resample(...)` where incorrect warnings were triggered by IPython introspection (GH13618)
- Bug in `NaT - Period` raises `AttributeError` (GH13071)
- Bug in Series comparison may output incorrect result if rhs contains `NaT` (GH9005)
- Bug in Series and Index comparison may output incorrect result if it contains `NaT` with object dtype (GH13592)
- Bug in Period addition raises `TypeError` if Period is on right hand side (GH13069)
- Bug in Period and Series or Index comparison raises `TypeError` (GH13200)
- Bug in `pd.set_eng_float_format()` that would prevent `NaN` and `Inf` from formatting (GH11981)
- Bug in `.unstack` with Categorical dtype resets `.ordered` to `True` (GH13249)
- Clean some compile time warnings in datetime parsing (GH13607)
- Bug in `factorize` raises `AmbiguousTimeError` if data contains datetime near DST boundary (GH13750)
- Bug in `.set_index` raises `AmbiguousTimeError` if new index contains DST boundary and multi levels (GH12920)
- Bug in `.shift` raises `AmbiguousTimeError` if data contains datetime near DST boundary (GH13926)
- Bug in `pd.read_hdf()` returns incorrect result when a DataFrame with a categorical column and a query which doesn't match any values (GH13792)
- Bug in `.iloc` when indexing with a non lex-sorted MultiIndex (GH13797)
- Bug in `.loc` when indexing with date strings in a reverse sorted DatetimeIndex (GH14316)
- Bug in Series comparison operators when dealing with zero dim NumPy arrays (GH13006)
- Bug in `.combine_first` may return incorrect dtype (GH7630, GH10567)
- Bug in groupby where `apply` returns different result depending on whether first result is `None` or not (GH12824)
- Bug in `groupby(...).nth()` where the group key is included inconsistently if called after `.head()` / `.tail()` (GH12839)
- Bug in `.to_html`, `.to_latex` and `.to_string` silently ignore custom datetime formatter passed through the `formatters` key word (GH10690)
- Bug in `DataFrame.iterrows()`, not yielding a Series subclasse if defined (GH13977)
- Bug in `pd.to_numeric` when `errors='coerce'` and input contains non-hashable objects (GH13324)
- Bug in invalid Timedelta arithmetic and comparison may raise `ValueError` rather than `TypeError` (GH13624)
- Bug in invalid datetime parsing in `to_datetime` and `DatetimeIndex` may raise `TypeError` rather than `ValueError` (GH11169, GH11287)
- Bug in Index created with tz-aware Timestamp and mismatched `tz` option incorrectly coerces timezone (GH13692)



- Bug in `DatetimeIndex` with nanosecond frequency does not include timestamp specified with `end` ([GH13672](#))
- Bug in `Series` when setting a slice with a `np.timedelta64` ([GH14155](#))
- Bug in `Index` raises `OutOfBoundsDatetime` if datetime exceeds `datetime64[ns]` bounds, rather than coercing to object dtype ([GH13663](#))
- Bug in `Index` may ignore specified `datetime64` or `timedelta64` passed as dtype ([GH13981](#))
- Bug in `RangeIndex` can be created without no arguments rather than raises `TypeError` ([GH13793](#))
- Bug in `.value_counts()` raises `OutOfBoundsDatetime` if data exceeds `datetime64[ns]` bounds ([GH13663](#))
- Bug in `DatetimeIndex` may raise `OutOfBoundsDatetime` if input `np.datetime64` has other unit than `ns` ([GH9114](#))
- Bug in `Series` creation with `np.datetime64` which has other unit than `ns` as object dtype results in incorrect values ([GH13876](#))
- Bug in `resample` with `timedelta` data where data was casted to float ([GH13119](#)).
- Bug in `pd.isnull()` `pd.notnull()` raise `TypeError` if input datetime-like has other unit than `ns` ([GH13389](#))
- Bug in `pd.merge()` may raise `TypeError` if input datetime-like has other unit than `ns` ([GH13389](#))
- Bug in `HDFStore/read_hdf()` discarded `DatetimeIndex.name` if `tz` was set ([GH13884](#))
- Bug in `Categorical.remove_unused_categories()` changes `.codes` dtype to platform int ([GH13261](#))
- Bug in `groupby` with `as_index=False` returns all NaN's when grouping on multiple columns including a categorical one ([GH13204](#))
- Bug in `df.groupby(...)[...]` where `getitem` with `Int64Index` raised an error ([GH13731](#))
- Bug in the CSS classes assigned to `DataFrame.style` for index names. Previously they were assigned `"col_heading level<n> col<c>"` where `n` was the number of levels + 1. Now they are assigned `"index_name level<n>"`, where `n` is the correct level for that `MultiIndex`.
- Bug where `pd.read_gbq()` could throw `ImportError`: No module named `discovery` as a result of a naming conflict with another python package called `apiclient` ([GH13454](#))
- Bug in `Index.union` returns an incorrect result with a named empty index ([GH13432](#))
- Bugs in `Index.difference` and `DataFrame.join` raise in Python3 when using mixed-integer indexes ([GH13432](#), [GH12814](#))
- Bug in subtract tz-aware `datetime.datetime` from tz-aware `datetime64` series ([GH14088](#))
- Bug in `.to_excel()` when `DataFrame` contains a `MultiIndex` which contains a label with a NaN value ([GH13511](#))
- Bug in invalid frequency offset string like `"D1"`, `"-2-3H"` may not raise `ValueError` ([GH13930](#))
- Bug in `concat` and `groupby` for hierarchical frames with `RangeIndex` levels ([GH13542](#)).
- Bug in `Series.str.contains()` for `Series` containing only NaN values of object dtype ([GH14171](#))
- Bug in `agg()` function on `groupby` dataframe changes dtype of `datetime64[ns]` column to `float64` ([GH12821](#))

- Bug in using NumPy ufunc with PeriodIndex to add or subtract integer raise IncompatibleFrequency. Note that using standard operator like + or - is recommended, because standard operators use more efficient path ([GH13980](#))
- Bug in operations on NaT returning float instead of datetime64[ns] ([GH12941](#))
- Bug in Series flexible arithmetic methods (like .add()) raises ValueError when axis=None ([GH13894](#))
- Bug in DataFrame.to\_csv() with MultiIndex columns in which a stray empty line was added ([GH6618](#))
- Bug in DatetimeIndex, TimedeltaIndex and PeriodIndex.equals() may return True when input isn't Index but contains the same values ([GH13107](#))
- Bug in assignment against datetime with timezone may not work if it contains datetime near DST boundary ([GH14146](#))
- Bug in pd.eval() and HDFStore query truncating long float literals with python 2 ([GH14241](#))
- Bug in Index raises KeyError displaying incorrect column when column is not in the df and columns contains duplicate values ([GH13822](#))
- Bug in Period and PeriodIndex creating wrong dates when frequency has combined offset aliases ([GH13874](#))
- Bug in .to\_string() when called with an integer line\_width and index=False raises an UnboundLocalError exception because idx referenced before assignment.
- Bug in eval() where the resolvers argument would not accept a list ([GH14095](#))
- Bugs in stack, get\_dummies, make\_axis\_dummies which don't preserve categorical dtypes in (multi)indexes ([GH13854](#))
- PeriodIndex can now accept list and array which contains pd.NaT ([GH13430](#))
- Bug in df.groupby where .median() returns arbitrary values if grouped dataframe contains empty bins ([GH13629](#))
- Bug in Index.copy() where name parameter was ignored ([GH14302](#))

## 1.13 v0.18.1 (May 3, 2016)

This is a minor bug-fix release from 0.18.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- .groupby(...) has been enhanced to provide convenient syntax when working with .rolling(...), .expanding(...) and .resample(...) per group, see [here](#)
- pd.to\_datetime() has gained the ability to assemble dates from a DataFrame, see [here](#)
- Method chaining improvements, see [here](#).
- Custom business hour offset, see [here](#).
- Many bug fixes in the handling of sparse, see [here](#)
- Expanded the *Tutorials section* with a feature on modern pandas, courtesy of [@TomAugsburger](#). ([GH13045](#)).



**What's new in v0.18.1**

- *New features*
  - *Custom Business Hour*
  - *.groupby(..) syntax with window and resample operations*
  - *Method chaining improvements*
    - \* *.where() and .mask()*
    - \* *.loc[], .iloc[], .ix[]*
    - \* *[] indexing*
  - *Partial string indexing on DateTimeIndex when part of a MultiIndex*
  - *Assembling Datetimes*
  - *Other Enhancements*
- *Sparse changes*
- *API changes*
  - *.groupby(..).nth() changes*
  - *numpy function compatibility*
  - *Using .apply on groupby resampling*
  - *Changes in read\_csv exceptions*
  - *to\_datetime error changes*
  - *Other API changes*
  - *Deprecations*
- *Performance Improvements*
- *Bug Fixes*

**1.13.1 New features****1.13.1.1 Custom Business Hour**

The CustomBusinessHour is a mixture of BusinessHour and CustomBusinessDay which allows you to specify arbitrary holidays. For details, see *Custom Business Hour* (GH11514)

```
In [1]: from pandas.tseries.offsets import CustomBusinessHour
In [2]: from pandas.tseries.holiday import USFederalHolidayCalendar
In [3]: bhour_us = CustomBusinessHour(calendar=USFederalHolidayCalendar())
```

Friday before MLK Day

```
In [4]: dt = datetime(2014, 1, 17, 15)
```

(continues on next page)

(continued from previous page)

```
In [5]: dt + bhour_us
Out[5]: Timestamp('2014-01-17 16:00:00')
```

Tuesday after MLK Day (Monday is skipped because it's a holiday)

```
In [6]: dt + bhour_us * 2
Out[6]: Timestamp('2014-01-20 09:00:00')
```

### 1.13.1.2 .groupby(...) syntax with window and resample operations

.groupby(...) has been enhanced to provide convenient syntax when working with .rolling(...), .expanding(...) and .resample(...) per group, see (GH12486, GH12738).

You can now use .rolling(...) and .expanding(...) as methods on groupbys. These return another deferred object (similar to what .rolling() and .expanding() do on ungrouped pandas objects). You can then operate on these RollingGroupby objects in a similar manner.

Previously you would have to do this to get a rolling window mean per-group:

```
In [7]: df = pd.DataFrame({'A': [1] * 20 + [2] * 12 + [3] * 8,
...:                      'B': np.arange(40)})
...:

In [8]: df
Out[8]:
```

	A	B
0	1	0
1	1	1
2	1	2
3	1	3
4	1	4
5	1	5
6	1	6
..	..	..
33	3	33
34	3	34
35	3	35
36	3	36
37	3	37
38	3	38
39	3	39

[40 rows x 2 columns]

```
In [9]: df.groupby('A').apply(lambda x: x.rolling(4).B.mean())
Out[9]:
```

A		
1	0	NaN
	1	NaN
	2	NaN
	3	1.5
	4	2.5
	5	3.5
	6	4.5
	...	...

(continues on next page)

(continued from previous page)

```

3  33      NaN
   34      NaN
   35     33.5
   36     34.5
   37     35.5
   38     36.5
   39     37.5
Name: B, Length: 40, dtype: float64

```

Now you can do:

```

In [10]: df.groupby('A').rolling(4).B.mean()
Out[10]:
A
1  0      NaN
   1      NaN
   2      NaN
   3      1.5
   4      2.5
   5      3.5
   6      4.5
...
3  33      NaN
   34      NaN
   35     33.5
   36     34.5
   37     35.5
   38     36.5
   39     37.5
Name: B, Length: 40, dtype: float64

```

For `.resample(...)` type of operations, previously you would have to:

```

In [11]: df = pd.DataFrame({'date': pd.date_range(start='2016-01-01',
.....:                                           periods=4,
.....:                                           freq='W'),
.....:                      'group': [1, 1, 2, 2],
.....:                      'val': [5, 6, 7, 8]}).set_index('date')
.....:

In [12]: df
Out[12]:
           group  val
date
2016-01-03      1    5
2016-01-10      1    6
2016-01-17      2    7
2016-01-24      2    8

```

```

In [13]: df.groupby('group').apply(lambda x: x.resample('1D').ffill())
Out[13]:
           group  val
group date
1  2016-01-03      1    5
   2016-01-04      1    5
   2016-01-05      1    5

```

(continues on next page)

(continued from previous page)

```

2016-01-06      1      5
2016-01-07      1      5
2016-01-08      1      5
2016-01-09      1      5
...
2    2016-01-18      2      7
    2016-01-19      2      7
    2016-01-20      2      7
    2016-01-21      2      7
    2016-01-22      2      7
    2016-01-23      2      7
    2016-01-24      2      8

[16 rows x 2 columns]
```

Now you can do:

```

In [14]: df.groupby('group').resample('1D').ffill()
Out[14]:
```

		group	val
group	date		
1	2016-01-03	1	5
	2016-01-04	1	5
	2016-01-05	1	5
	2016-01-06	1	5
	2016-01-07	1	5
	2016-01-08	1	5
	2016-01-09	1	5
...		...	...
2	2016-01-18	2	7
	2016-01-19	2	7
	2016-01-20	2	7
	2016-01-21	2	7
	2016-01-22	2	7
	2016-01-23	2	7
	2016-01-24	2	8

```

[16 rows x 2 columns]
```

### 1.13.1.3 Method chaining improvements

The following methods / indexers now accept a callable. It is intended to make these more useful in method chains, see the [documentation](#). (GH11485, GH12533)

- `.where()` and `.mask()`
- `.loc[], iloc[]` and `.ix[]`
- `[]` indexing

#### `.where()` and `.mask()`

These can accept a callable for the condition and other arguments.

```
In [15]: df = pd.DataFrame({'A': [1, 2, 3],
.....:                    'B': [4, 5, 6],
.....:                    'C': [7, 8, 9]})
.....:

In [16]: df.where(lambda x: x > 4, lambda x: x + 10)
Out[16]:
```

	A	B	C
0	11	14	7
1	12	5	8
2	13	6	9

`.loc[], .iloc[], .ix[]`

These can accept a callable, and a tuple of callable as a slicer. The callable can return a valid boolean indexer or anything which is valid for these indexer's input.

```
# callable returns bool indexer
In [17]: df.loc[lambda x: x.A >= 2, lambda x: x.sum() > 10]
Out[17]:
```

	B	C
1	5	8
2	6	9

```
# callable returns list of labels
In [18]: df.loc[lambda x: [1, 2], lambda x: ['A', 'B']]
Out[18]:
```

	A	B
1	2	5
2	3	6

## [ ] indexing

Finally, you can use a callable in [ ] indexing of Series, DataFrame and Panel. The callable must return a valid input for [ ] indexing depending on its class and index type.

```
In [19]: df[lambda x: 'A']
Out[19]:
```

	A
0	1
1	2
2	3

Name: A, dtype: int64

Using these methods / indexers, you can chain data selection operations without using temporary variable.

```
In [20]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [21]: (bb.groupby(['year', 'team'])
.....:      .sum()
.....:      .loc[lambda df: df.r > 100]
.....:      )
.....:
Out[21]:
```

(continues on next page)

(continued from previous page)

		stint	g	ab	r	h	X2b	X3b	hr	rbi	sb	cs	bb	so	
↪ibb	hbp	sh	sf	gidp											↪
year	team														↪
↪															
2007	CIN	6	379	745	101	203	35	2	36	125.0	10.0	1.0	105	127.0	14.
↪0	1.0	1.0	15.0	18.0											
	DET	5	301	1062	162	283	54	4	37	144.0	24.0	7.0	97	176.0	3.
↪0	10.0	4.0	8.0	28.0											
	HOU	4	311	926	109	218	47	6	14	77.0	10.0	4.0	60	212.0	3.
↪0	9.0	16.0	6.0	17.0											
	LAN	11	413	1021	153	293	61	3	36	154.0	7.0	5.0	114	141.0	8.
↪0	9.0	3.0	8.0	29.0											
	NYN	13	622	1854	240	509	101	3	61	243.0	22.0	4.0	174	310.0	24.
↪0	23.0	18.0	15.0	48.0											
	SFN	5	482	1305	198	337	67	6	40	171.0	26.0	7.0	235	188.0	51.
↪0	8.0	16.0	6.0	41.0											
	TEX	2	198	729	115	200	40	4	28	115.0	21.0	4.0	73	140.0	4.
↪0	5.0	2.0	8.0	16.0											
	TOR	4	459	1408	187	378	96	2	58	223.0	4.0	2.0	190	265.0	16.
↪0	12.0	4.0	16.0	38.0											

#### 1.13.1.4 Partial string indexing on DateTimeIndex when part of a MultiIndex

Partial string indexing now matches on DateTimeIndex when part of a MultiIndex (GH10331)

```
In [22]: dft2 = pd.DataFrame(np.random.randn(20, 1),
.....:                      columns=['A'],
.....:                      index=pd.MultiIndex.from_product([pd.date_range('20130101
↪',
.....:
↪periods=10,
.....:
↪),
.....:                      freq='12H
.....:                      ['a', 'b']]))
```

In [23]: dft2

Out [23]:

```

      A
2013-01-01 00:00:00 a  0.156998
                  b -0.571455
2013-01-01 12:00:00 a  1.057633
                  b -0.791489
2013-01-02 00:00:00 a -0.524627
                  b  0.071878
2013-01-02 12:00:00 a  1.910759
...
2013-01-04 00:00:00 b  1.015405
2013-01-04 12:00:00 a  0.749185
                  b -0.675521
2013-01-05 00:00:00 a  0.440266
                  b  0.688972
2013-01-05 12:00:00 a -0.276646
                  b  1.924533
```

(continues on next page)

(continued from previous page)

[20 rows x 1 columns]

**In [24]:** dft2.loc['2013-01-05']

```

////////////////////////////////////
↪
                                     A
2013-01-05 00:00:00 a    0.440266
                                     b    0.688972
2013-01-05 12:00:00 a   -0.276646
                                     b    1.924533

```

On other levels

**In [25]:** idx = pd.IndexSlice**In [26]:** dft2 = dft2.swaplevel(0, 1).sort\_index()**In [27]:** dft2**Out[27]:**

```

                                     A
a 2013-01-01 00:00:00    0.156998
   2013-01-01 12:00:00    1.057633
   2013-01-02 00:00:00   -0.524627
   2013-01-02 12:00:00    1.910759
   2013-01-03 00:00:00    0.513082
   2013-01-03 12:00:00    1.043945
   2013-01-04 00:00:00    1.459927
...
b 2013-01-02 12:00:00    0.787965
   2013-01-03 00:00:00   -0.546416
   2013-01-03 12:00:00    2.107785
   2013-01-04 00:00:00    1.015405
   2013-01-04 12:00:00   -0.675521
   2013-01-05 00:00:00    0.688972
   2013-01-05 12:00:00    1.924533

```

[20 rows x 1 columns]

**In [28]:** dft2.loc[idx[:, '2013-01-05'], :]

```

////////////////////////////////////
↪
                                     A
a 2013-01-05 00:00:00    0.440266
   2013-01-05 12:00:00   -0.276646
b 2013-01-05 00:00:00    0.688972
   2013-01-05 12:00:00    1.924533

```

### 1.13.1.5 Assembling Datetimes

`pd.to_datetime()` has gained the ability to assemble datetimes from a passed in `DataFrame` or a dict. (GH8158).

```

In [29]: df = pd.DataFrame({'year': [2015, 2016],
    ....:                    'month': [2, 3],
    ....:                    'day': [4, 5],

```

(continues on next page)

(continued from previous page)

```

.....:             'hour': [2, 3])
.....:
In [30]: df
Out[30]:
   year  month  day  hour
0  2015     2    4     2
1  2016     3    5     3

```

Assembling using the passed frame.

```

In [31]: pd.to_datetime(df)
Out[31]:
0    2015-02-04 02:00:00
1    2016-03-05 03:00:00
dtype: datetime64[ns]

```

You can pass only the columns that you need to assemble.

```

In [32]: pd.to_datetime(df[['year', 'month', 'day']])
Out[32]:
0    2015-02-04
1    2016-03-05
dtype: datetime64[ns]

```

### 1.13.1.6 Other Enhancements

- `pd.read_csv()` now supports `delim_whitespace=True` for the Python engine ([GH12958](#))
- `pd.read_csv()` now supports opening ZIP files that contains a single CSV, via extension inference or explicit `compression='zip'` ([GH12175](#))
- `pd.read_csv()` now supports opening files using xz compression, via extension inference or explicit `compression='xz'` is specified; xz compressions is also supported by `DataFrame.to_csv` in the same way ([GH11852](#))
- `pd.read_msgpack()` now always gives writeable ndarrays even when compression is used ([GH12359](#)).
- `pd.read_msgpack()` now supports serializing and de-serializing categoricals with msgpack ([GH12573](#))
- `.to_json()` now supports NDFrames that contain categorical and sparse data ([GH10778](#))
- `interpolate()` now supports `method='akima'` ([GH7588](#)).
- `pd.read_excel()` now accepts path objects (e.g. `pathlib.Path`, `py.path.local`) for the file path, in line with other `read_*` functions ([GH12655](#))
- Added `.weekday_name` property as a component to `DatetimeIndex` and the `.dt` accessor. ([GH11128](#))
- `Index.take` now handles `allow_fill` and `fill_value` consistently ([GH12631](#))

```

In [33]: idx = pd.Index([1., 2., 3., 4.], dtype='float')

# default, allow_fill=True, fill_value=None
In [34]: idx.take([2, -1])
Out[34]: Float64Index([3.0, 4.0], dtype='float64')

```

(continues on next page)



(continued from previous page)

```
In [35]: idx.take([2, -1], fill_value=True)
Out[35]: Float64Index([3.0,
nan], dtype='float64')
```

- Index now supports `.str.get_dummies()` which returns `MultiIndex`, see *Creating Indicator Variables* (GH10008, GH10103)

```
In [36]: idx = pd.Index(['a|b', 'a|c', 'b|c'])

In [37]: idx.str.get_dummies('|')
Out[37]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
            labels=[[1, 1, 0], [1, 0, 1], [0, 1, 1]],
            names=['a', 'b', 'c'])
```

- `pd.crosstab()` has gained a `normalize` argument for normalizing frequency tables (GH12569). Examples in the updated docs [here](#).
- `.resample(...).interpolate()` is now supported (GH12925)
- `.isin()` now accepts passed sets (GH12988)

### 1.13.2 Sparse changes

These changes conform sparse handling to return the correct types and work to make a smoother experience with indexing.

`SparseArray.take` now returns a scalar for scalar input, `SparseArray` for others. Furthermore, it handles a negative indexer with the same rule as `Index` ([GH10560](#), [GH12796](#))

```
In [38]: s = pd.SparseArray([np.nan, np.nan, 1, 2, 3, np.nan, 4, 5, np.nan, 6])

In [39]: s.take(0)
Out[39]: nan

In [40]: s.take([1, 2, 3])
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[40]:
[nan, 1.0, 2.0]
Fill: nan
IntIndex
Indices: array([1, 2], dtype=int32)
```

- Bug in `SparseSeries[]` indexing with Ellipsis raises `KeyError` ([GH9467](#))
- Bug in `SparseArray[]` indexing with tuples are not handled properly ([GH12966](#))
- Bug in `SparseSeries.loc[]` with list-like input raises `TypeError` ([GH10560](#))
- Bug in `SparseSeries.iloc[]` with scalar input may raise `IndexError` ([GH10560](#))
- Bug in `SparseSeries.loc[], .iloc[]` with slice returns `SparseArray`, rather than `SparseSeries` ([GH10560](#))
- Bug in `SparseDataFrame.loc[], .iloc[]` may results in dense `Series`, rather than `SparseSeries` ([GH12787](#))
- Bug in `SparseArray` addition ignores `fill_value` of right hand side ([GH12910](#))
- Bug in `SparseArray` mod raises `AttributeError` ([GH12910](#))

- Bug in `SparseArray` `pow` calculates `1 ** np.nan` as `np.nan` which must be 1 (GH12910)
- Bug in `SparseArray` comparison output may incorrect result or raise `ValueError` (GH12971)
- Bug in `SparseSeries.__repr__` raises `TypeError` when it is longer than `max_rows` (GH10560)
- Bug in `SparseSeries.shape` ignores `fill_value` (GH10452)
- Bug in `SparseSeries` and `SparseArray` may have different `dtype` from its dense values (GH12908)
- Bug in `SparseSeries.reindex` incorrectly handle `fill_value` (GH12797)
- Bug in `SparseArray.to_frame()` results in `DataFrame`, rather than `SparseDataFrame` (GH9850)
- Bug in `SparseSeries.value_counts()` does not count `fill_value` (GH6749)
- Bug in `SparseArray.to_dense()` does not preserve `dtype` (GH10648)
- Bug in `SparseArray.to_dense()` incorrectly handle `fill_value` (GH12797)
- Bug in `pd.concat()` of `SparseSeries` results in dense (GH10536)
- Bug in `pd.concat()` of `SparseDataFrame` incorrectly handle `fill_value` (GH9765)
- Bug in `pd.concat()` of `SparseDataFrame` may raise `AttributeError` (GH12174)
- Bug in `SparseArray.shift()` may raise `NameError` or `TypeError` (GH12908)

## 1.13.3 API changes

### 1.13.3.1 `.groupby(...).nth()` changes

The index in `.groupby(...).nth()` output is now more consistent when the `as_index` argument is passed (GH11039):

```
In [41]: df = DataFrame({'A' : ['a', 'b', 'a'],
.....:                  'B' : [1, 2, 3]})
.....:

In [42]: df
Out[42]:
   A  B
0  a  1
1  b  2
2  a  3
```

Previous Behavior:

```
In [3]: df.groupby('A', as_index=True)['B'].nth(0)
Out[3]:
0    1
1    2
Name: B, dtype: int64

In [4]: df.groupby('A', as_index=False)['B'].nth(0)
Out[4]:
0    1
1    2
Name: B, dtype: int64
```

New Behavior:

```
In [43]: df.groupby('A', as_index=True)['B'].nth(0)
Out[43]:
A
a    1
b    2
Name: B, dtype: int64

In [44]: df.groupby('A', as_index=False)['B'].nth(0)
Out[44]:
0    1
1    2
Name: B, dtype: int64
```

Furthermore, previously, a `.groupby` would always sort, regardless if `sort=False` was passed with `.nth()`.

```
In [45]: np.random.seed(1234)

In [46]: df = pd.DataFrame(np.random.randn(100, 2), columns=['a', 'b'])

In [47]: df['c'] = np.random.randint(0, 4, 100)
```

Previous Behavior:

```
In [4]: df.groupby('c', sort=True).nth(1)
Out[4]:
      a      b
c
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524

In [5]: df.groupby('c', sort=False).nth(1)
Out[5]:
      a      b
c
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524
```

New Behavior:

```
In [48]: df.groupby('c', sort=True).nth(1)
Out[48]:
      a      b
c
0 -0.334077  0.002118
1  0.036142 -2.074978
2 -0.720589  0.887163
3  0.859588 -0.636524

In [49]: df.groupby('c', sort=False).nth(1)
Out[49]:
      a      b
c
```

(continues on next page)

(continued from previous page)

```

2 -0.720589  0.887163
3  0.859588 -0.636524
0 -0.334077  0.002118
1  0.036142 -2.074978

```

### 1.13.3.2 numpy function compatibility

Compatibility between pandas array-like methods (e.g. `sum` and `take`) and their `numpy` counterparts has been greatly increased by augmenting the signatures of the pandas methods so as to accept arguments that can be passed in from `numpy`, even if they are not necessarily used in the pandas implementation ([GH12644](#), [GH12638](#), [GH12687](#))

- `.searchsorted()` for `Index` and `TimedeltaIndex` now accept a `sorter` argument to maintain compatibility with `numpy`'s `searchsorted` function ([GH12238](#))
- Bug in `numpy` compatibility of `np.round()` on a `Series` ([GH12600](#))

An example of this signature augmentation is illustrated below:

```

In [50]: sp = pd.SparseDataFrame([1, 2, 3])

In [51]: sp
Out[51]:
   0
0  1
1  2
2  3

```

Previous behaviour:

```

In [2]: np.cumsum(sp, axis=0)
...
TypeError: cumsum() takes at most 2 arguments (4 given)

```

New behaviour:

```

In [52]: np.cumsum(sp, axis=0)
Out[52]:
   0
0  1
1  3
2  6

```

### 1.13.3.3 Using `.apply` on groupby resampling

Using `apply` on resampling groupby operations (using a `pd.TimeGrouper`) now has the same output types as similar `apply` calls on other groupby operations. ([GH11742](#)).

```

In [53]: df = pd.DataFrame({'date': pd.to_datetime(['10/10/2000', '11/10/2000']),
...:                        'value': [10, 13]})
...:
...:

In [54]: df
Out[54]:
   date  value
0 10/10/2000    10
1 11/10/2000    13

```

(continues on next page)

(continued from previous page)

```
0 2000-10-10    10
1 2000-11-10    13
```

Previous behavior:

```
In [1]: df.groupby(pd.TimeGrouper(key='date', freq='M')).apply(lambda x: x.value.
↳sum())
Out[1]:
...
TypeError: cannot concatenate a non-NDFrame object

# Output is a Series
In [2]: df.groupby(pd.TimeGrouper(key='date', freq='M')).apply(lambda x: x[['value']].
↳sum())
Out[2]:
date
2000-10-31  value    10
2000-11-30  value    13
dtype: int64
```

New Behavior:

```
# Output is a Series
In [55]: df.groupby(pd.TimeGrouper(key='date', freq='M')).apply(lambda x: x.value.
↳sum())
Out[55]:
date
2000-10-31    10
2000-11-30    13
Freq: M, dtype: int64

# Output is a DataFrame
In [56]: df.groupby(pd.TimeGrouper(key='date', freq='M')).apply(lambda x: x[['value
↳']].sum())
Out[56]:
      value
date
2000-10-31    10
2000-11-30    13
```

### 1.13.3.4 Changes in read\_csv exceptions

In order to standardize the `read_csv` API for both the `c` and `python` engines, both will now raise an `EmptyDataError`, a subclass of `ValueError`, in response to empty columns or header ([GH12493](#), [GH12506](#))

Previous behaviour:

```
In [1]: df = pd.read_csv(StringIO(''), engine='c')
...
ValueError: No columns to parse from file

In [2]: df = pd.read_csv(StringIO(''), engine='python')
...
StopIteration
```

New behaviour:

```
In [1]: df = pd.read_csv(StringIO(''), engine='c')
...
pandas.io.common.EmptyDataError: No columns to parse from file

In [2]: df = pd.read_csv(StringIO(''), engine='python')
...
pandas.io.common.EmptyDataError: No columns to parse from file
```

In addition to this error change, several others have been made as well:

- CParserError now sub-classes ValueError instead of just a Exception ([GH12551](#))
- A CParserError is now raised instead of a generic Exception in read\_csv when the c engine cannot parse a column ([GH12506](#))
- A ValueError is now raised instead of a generic Exception in read\_csv when the c engine encounters a NaN value in an integer column ([GH12506](#))
- A ValueError is now raised instead of a generic Exception in read\_csv when true\_values is specified, and the c engine encounters an element in a column containing unencodable bytes ([GH12506](#))
- pandas.parser.OverflowError exception has been removed and has been replaced with Python's built-in OverflowError exception ([GH12506](#))
- pd.read\_csv() no longer allows a combination of strings and integers for the usecols parameter ([GH12678](#))

### 1.13.3.5 to\_datetime error changes

Bugs in pd.to\_datetime() when passing a unit with convertible entries and errors='coerce' or non-convertible with errors='ignore'. Furthermore, an OutOfBoundsDatetime exception will be raised when an out-of-range value is encountered for that unit when errors='raise'. ([GH11758](#), [GH13052](#), [GH13059](#))

Previous behaviour:

```
In [27]: pd.to_datetime(1420043460, unit='s', errors='coerce')
Out[27]: NaT

In [28]: pd.to_datetime(11111111, unit='D', errors='ignore')
OverflowError: Python int too large to convert to C long

In [29]: pd.to_datetime(11111111, unit='D', errors='raise')
OverflowError: Python int too large to convert to C long
```

New behaviour:

```
In [2]: pd.to_datetime(1420043460, unit='s', errors='coerce')
Out[2]: Timestamp('2014-12-31 16:31:00')

In [3]: pd.to_datetime(11111111, unit='D', errors='ignore')
Out[3]: 11111111

In [4]: pd.to_datetime(11111111, unit='D', errors='raise')
OutOfBoundsDatetime: cannot convert input with unit 'D'
```

### 1.13.3.6 Other API changes

- `.swaplevel()` for `Series`, `DataFrame`, `Panel`, and `MultiIndex` now features defaults for its first two parameters `i` and `j` that swap the two innermost levels of the index. (GH12934)
- `.searchsorted()` for `Index` and `TimedeltaIndex` now accept a `sorter` argument to maintain compatibility with `numpy`'s `searchsorted` function (GH12238)
- `Period` and `PeriodIndex` now raises `IncompatibleFrequency` error which inherits `ValueError` rather than raw `ValueError` (GH12615)
- `Series.apply` for category dtype now applies the passed function to each of the `.categories` (and not the `.codes`), and returns a category dtype if possible (GH12473)
- `read_csv` will now raise a `TypeError` if `parse_dates` is neither a boolean, list, or dictionary (matches the doc-string) (GH5636)
- The default for `.query()/eval()` is now `engine=None`, which will use `numexpr` if it's installed; otherwise it will fallback to the `python` engine. This mimics the pre-0.18.1 behavior if `numexpr` is installed (and which, previously, if `numexpr` was not installed, `.query()/eval()` would raise). (GH12749)
- `pd.show_versions()` now includes `pandas_datareader` version (GH12740)
- Provide a proper `__name__` and `__qualname__` attributes for generic functions (GH12021)
- `pd.concat(ignore_index=True)` now uses `RangeIndex` as default (GH12695)
- `pd.merge()` and `DataFrame.join()` will show a `UserWarning` when merging/joining a single- with a multi-leveled dataframe (GH9455, GH12219)
- Compat with `scipy > 0.17` for deprecated `piecewise_polynomial` interpolation method; support for the replacement `from_derivatives` method (GH12887)

### 1.13.3.7 Deprecations

- The method name `Index.sym_diff()` is deprecated and can be replaced by `Index.symmetric_difference()` (GH12591)
- The method name `Categorical.sort()` is deprecated in favor of `Categorical.sort_values()` (GH12882)

### 1.13.4 Performance Improvements

- Improved speed of SAS reader (GH12656, GH12961)
- Performance improvements in `.groupby(...).cumcount()` (GH11039)
- Improved memory usage in `pd.read_csv()` when using `skiprows=an_integer` (GH13005)
- Improved performance of `DataFrame.to_sql` when checking case sensitivity for tables. Now only checks if table has been created correctly when table name is not lower case. (GH12876)
- Improved performance of `Period` construction and time series plotting (GH12903, GH11831).
- Improved performance of `.str.encode()` and `.str.decode()` methods (GH13008)
- Improved performance of `to_numeric` if input is numeric dtype (GH12777)
- Improved performance of sparse arithmetic with `IntIndex` (GH13036)

### 1.13.5 Bug Fixes

- `usecols` parameter in `pd.read_csv` is now respected even when the lines of a CSV file are not even ([GH12203](#))
- Bug in `groupby.transform(...)` when `axis=1` is specified with a non-monotonic ordered index ([GH12713](#))
- Bug in `Period` and `PeriodIndex` creation raises `KeyError` if `freq="Minute"` is specified. Note that “Minute” `freq` is deprecated in v0.17.0, and recommended to use `freq="T"` instead ([GH11854](#))
- Bug in `.resample(...).count()` with a `PeriodIndex` always raising a `TypeError` ([GH12774](#))
- Bug in `.resample(...)` with a `PeriodIndex` casting to a `DatetimeIndex` when empty ([GH12868](#))
- Bug in `.resample(...)` with a `PeriodIndex` when resampling to an existing frequency ([GH12770](#))
- Bug in printing data which contains `Period` with different `freq` raises `ValueError` ([GH12615](#))
- Bug in `Series` construction with `Categorical` and `dtype='category'` is specified ([GH12574](#))
- Bugs in concatenation with a coercible `dtype` was too aggressive, resulting in different `dtypes` in output formatting when an object was longer than `display.max_rows` ([GH12411](#), [GH12045](#), [GH11594](#), [GH10571](#), [GH12211](#))
- Bug in `float_format` option with option not being validated as a callable. ([GH12706](#))
- Bug in `GroupBy.filter` when `dropna=False` and no groups fulfilled the criteria ([GH12768](#))
- Bug in `__name__` of `.cum*` functions ([GH12021](#))
- Bug in `.astype()` of a `Float64Index/Int64Index` to an `Int64Index` ([GH12881](#))
- Bug in roundtripping an integer based index in `.to_json()/read_json()` when `orient='index'` (the default) ([GH12866](#))
- Bug in plotting `Categorical` `dtypes` cause error when attempting stacked bar plot ([GH13019](#))
- Compat with `>= numpy 1.11` for `NaT` comparisons ([GH12969](#))
- Bug in `.drop()` with a non-unique `MultiIndex`. ([GH12701](#))
- Bug in `.concat` of datetime tz-aware and naive `DataFrames` ([GH12467](#))
- Bug in correctly raising a `ValueError` in `.resample(...).fillna(...)` when passing a non-string ([GH12952](#))
- Bug fixes in various encoding and header processing issues in `pd.read_sas()` ([GH12659](#), [GH12654](#), [GH12647](#), [GH12809](#))
- Bug in `pd.crosstab()` where would silently ignore `aggfunc` if `values=None` ([GH12569](#)).
- Potential segfault in `DataFrame.to_json` when serialising `datetime.time` ([GH11473](#)).
- Potential segfault in `DataFrame.to_json` when attempting to serialise `0d` array ([GH11299](#)).
- Segfault in `to_json` when attempting to serialise a `DataFrame` or `Series` with non-ndarray values; now supports serialization of `category`, `sparse`, and `datetime64[ns, tz]` `dtypes` ([GH10778](#)).
- Bug in `DataFrame.to_json` with unsupported `dtype` not passed to default handler ([GH12554](#)).
- Bug in `.align` not returning the sub-class ([GH12983](#))
- Bug in aligning a `Series` with a `DataFrame` ([GH13037](#))
- Bug in `ABCPanel` in which `Panel4D` was not being considered as a valid instance of this generic type ([GH12810](#))



- Bug in consistency of `.name` on `.groupby(..).apply(..)` cases ([GH12363](#))
- Bug in `Timestamp.__repr__` that caused `pprint` to fail in nested structures ([GH12622](#))
- Bug in `Timedelta.min` and `Timedelta.max`, the properties now report the true minimum/maximum `timedeltas` as recognized by pandas. See the [documentation](#). ([GH12727](#))
- Bug in `.quantile()` with interpolation may coerce to `float` unexpectedly ([GH12772](#))
- Bug in `.quantile()` with empty `Series` may return scalar rather than empty `Series` ([GH12772](#))
- Bug in `.loc` with out-of-bounds in a large indexer would raise `IndexError` rather than `KeyError` ([GH12527](#))
- Bug in resampling when using a `TimedeltaIndex` and `.asfreq()`, would previously not include the final fencepost ([GH12926](#))
- Bug in equality testing with a `Categorical` in a `DataFrame` ([GH12564](#))
- Bug in `GroupBy.first()`, `.last()` returns incorrect row when `TimeGrouper` is used ([GH7453](#))
- Bug in `pd.read_csv()` with the `c` engine when specifying `skiprows` with newlines in quoted items ([GH10911](#), [GH12775](#))
- Bug in `DataFrame` `timezone` lost when assigning `tz-aware` `datetime` `Series` with `alignment` ([GH12981](#))
- Bug in `.value_counts()` when `normalize=True` and `dropna=True` where `nulls` still contributed to the normalized count ([GH12558](#))
- Bug in `Series.value_counts()` loses `name` if its `dtype` is `category` ([GH12835](#))
- Bug in `Series.value_counts()` loses `timezone` info ([GH12835](#))
- Bug in `Series.value_counts(normalize=True)` with `Categorical` raises `UnboundLocalError` ([GH12835](#))
- Bug in `Panel.fillna()` ignoring `inplace=True` ([GH12633](#))
- Bug in `pd.read_csv()` when specifying `names`, `usecols`, and `parse_dates` simultaneously with the `c` engine ([GH9755](#))
- Bug in `pd.read_csv()` when specifying `delim_whitespace=True` and `lineterminator` simultaneously with the `c` engine ([GH12912](#))
- Bug in `Series.rename`, `DataFrame.rename` and `DataFrame.rename_axis` not treating `Series` as mappings to relabel ([GH12623](#)).
- Clean in `.rolling.min` and `.rolling.max` to enhance `dtype` handling ([GH12373](#))
- Bug in `groupby` where complex types are coerced to `float` ([GH12902](#))
- Bug in `Series.map` raises `TypeError` if its `dtype` is `category` or `tz-aware` `datetime` ([GH12473](#))
- Bugs on 32bit platforms for some test comparisons ([GH12972](#))
- Bug in index coercion when falling back from `RangeIndex` construction ([GH12893](#))
- Better error message in window functions when invalid argument (e.g. a `float` window) is passed ([GH12669](#))
- Bug in slicing subclassed `DataFrame` defined to return subclassed `Series` may return normal `Series` ([GH11559](#))
- Bug in `.str` accessor methods may raise `ValueError` if input has `name` and the result is `DataFrame` or `MultiIndex` ([GH12617](#))
- Bug in `DataFrame.last_valid_index()` and `DataFrame.first_valid_index()` on empty frames ([GH12800](#))

- Bug in `CategoricalIndex.get_loc` returns different result from regular `Index` ([GH12531](#))
- Bug in `PeriodIndex.resample` where name not propagated ([GH12769](#))
- Bug in `date_range` closed keyword and timezones ([GH12684](#)).
- Bug in `pd.concat` raises `AttributeError` when input data contains tz-aware datetime and timedelta ([GH12620](#))
- Bug in `pd.concat` did not handle empty `Series` properly ([GH11082](#))
- Bug in `.plot.bar` alignment when width is specified with `int` ([GH12979](#))
- Bug in `fill_value` is ignored if the argument to a binary operator is a constant ([GH12723](#))
- Bug in `pd.read_html()` when using `bs4` flavor and parsing table with a header and only one column ([GH9178](#))
- Bug in `.pivot_table` when `margins=True` and `dropna=True` where nulls still contributed to margin count ([GH12577](#))
- Bug in `.pivot_table` when `dropna=False` where table index/column names disappear ([GH12133](#))
- Bug in `pd.crosstab()` when `margins=True` and `dropna=False` which raised ([GH12642](#))
- Bug in `Series.name` when name attribute can be a hashable type ([GH12610](#))
- Bug in `.describe()` resets categorical columns information ([GH11558](#))
- Bug where `loffset` argument was not applied when calling `resample().count()` on a timeseries ([GH12725](#))
- `pd.read_excel()` now accepts column names associated with keyword argument `names` ([GH12870](#))
- Bug in `pd.to_numeric()` with `Index` returns `np.ndarray`, rather than `Index` ([GH12777](#))
- Bug in `pd.to_numeric()` with datetime-like may raise `TypeError` ([GH12777](#))
- Bug in `pd.to_numeric()` with scalar raises `ValueError` ([GH12777](#))

## 1.14 v0.18.0 (March 13, 2016)

This is a major release from 0.17.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

**Warning:** pandas >= 0.18.0 no longer supports compatibility with Python version 2.6 and 3.3 ([GH7718](#), [GH11273](#))

**Warning:** numexpr version 2.4.4 will now show a warning and not be used as a computation back-end for pandas because of some buggy behavior. This does not affect other versions (>= 2.1 and >= 2.4.6). ([GH12489](#))

Highlights include:

- Moving and expanding window functions are now methods on `Series` and `DataFrame`, similar to `.groupby`, see [here](#).
- Adding support for a `RangeIndex` as a specialized form of the `Int64Index` for memory savings, see [here](#).

- API breaking change to the `.resample` method to make it more `.groupby` like, see [here](#).
- Removal of support for positional indexing with floats, which was deprecated since 0.14.0. This will now raise a `TypeError`, see [here](#).
- The `.to_xarray()` function has been added for compatibility with the `xarray` package, see [here](#).
- The `read_sas` function has been enhanced to read `sas7bdat` files, see [here](#).
- Addition of the `.str.extractall()` method, and API changes to the `.str.extract()` method and `.str.cat()` method.
- `pd.test()` top-level nose test runner is available ([GH4327](#)).

Check the [API Changes](#) and [deprecations](#) before updating.

### What's new in v0.18.0

- *New features*
  - *Window functions are now methods*
  - *Changes to rename*
  - *Range Index*
  - *Changes to str.extract*
  - *Addition of str.extractall*
  - *Changes to str.cat*
  - *Datetimelike rounding*
  - *Formatting of Integers in FloatIndex*
  - *Changes to dtype assignment behaviors*
  - *to\_xarray*
  - *Latex Representation*
  - *pd.read\_sas() changes*
  - *Other enhancements*
- *Backwards incompatible API changes*
  - *NaT and Timedelta operations*
  - *Changes to msgpack*
  - *Signature change for .rank*
  - *Bug in QuarterBegin with n=0*
  - *Resample API*
    - \* *Downsampling*
    - \* *Upsampling*
    - \* *Previous API will work but with deprecations*
  - *Changes to eval*
  - *Other API Changes*
  - *Deprecations*

- *Removal of deprecated float indexers*
- *Removal of prior version deprecations/changes*
- *Performance Improvements*
- *Bug Fixes*

## 1.14.1 New features

### 1.14.1.1 Window functions are now methods

Window functions have been refactored to be methods on `Series/DataFrame` objects, rather than top-level functions, which are now deprecated. This allows these window-type functions, to have a similar API to that of `.groupby`. See the full documentation [here](#) (GH11603, GH12373)

```
In [1]: np.random.seed(1234)

In [2]: df = pd.DataFrame({'A' : range(10), 'B' : np.random.randn(10)})

In [3]: df
Out[3]:
```

	A	B
0	0	0.471435
1	1	-1.190976
2	2	1.432707
3	3	-0.312652
4	4	-0.720589
5	5	0.887163
6	6	0.859588
7	7	-0.636524
8	8	0.015696
9	9	-2.242685

Previous Behavior:

```
In [8]: pd.rolling_mean(df,window=3)
FutureWarning: pd.rolling_mean is deprecated for DataFrame and will be
removed in a future version, replace with
DataFrame.rolling(window=3,center=False).mean()

Out[8]:
```

	A	B
0	NaN	NaN
1	NaN	NaN
2	1	0.237722
3	2	-0.023640
4	3	0.133155
5	4	-0.048693
6	5	0.342054
7	6	0.370076
8	7	0.079587
9	8	-0.954504

New Behavior:

```
In [4]: r = df.rolling(window=3)
```

These show a descriptive repr

```
In [5]: r
Out[5]: Rolling [window=3,center=False,axis=0]
```

with tab-completion of available methods and properties.

```
In [9]: r.
r.A          r.agg          r.apply          r.count          r.exclusions  r.max          r.
↳median      r.name          r.skew          r.sum
r.B          r.aggregate      r.corr          r.cov          r.kurt        r.mean        r.
↳min         r.quantile      r.std          r.var
```

The methods operate on the Rolling object itself

```
In [6]: r.mean()
Out[6]:
   A      B
0 NaN    NaN
1 NaN    NaN
2 1.0  0.237722
3 2.0 -0.023640
4 3.0  0.133155
5 4.0 -0.048693
6 5.0  0.342054
7 6.0  0.370076
8 7.0  0.079587
9 8.0 -0.954504
```

They provide getitem accessors

```
In [7]: r['A'].mean()
Out[7]:
0    NaN
1    NaN
2    1.0
3    2.0
4    3.0
5    4.0
6    5.0
7    6.0
8    7.0
9    8.0
Name: A, dtype: float64
```

And multiple aggregations

```
In [8]: r.agg({'A' : ['mean','std'],
...          'B' : ['mean','std']})
Out[8]:
   A      B
mean std mean std
0 NaN NaN NaN NaN
1 NaN NaN NaN NaN
2 1.0 1.0 0.237722 1.327364
3 2.0 1.0 -0.023640 1.335505
4 3.0 1.0 0.133155 1.143778
```

(continues on next page)

(continued from previous page)

```

5  4.0  1.0 -0.048693  0.835747
6  5.0  1.0  0.342054  0.920379
7  6.0  1.0  0.370076  0.871850
8  7.0  1.0  0.079587  0.750099
9  8.0  1.0 -0.954504  1.162285

```

### 1.14.1.2 Changes to rename

`Series.rename` and `NDFrame.rename_axis` can now take a scalar or list-like argument for altering the Series or axis *name*, in addition to their old behaviors of altering labels. ([GH9494](#), [GH11965](#))

```
In [9]: s = pd.Series(np.random.randn(5))
```

```
In [10]: s.rename('newname')
```

```
Out[10]:
```

```
0    1.150036
```

```
1    0.991946
```

```
2    0.953324
```

```
3   -2.021255
```

```
4   -0.334077
```

```
Name: newname, dtype: float64
```

```
In [11]: df = pd.DataFrame(np.random.randn(5, 2))
```

```
In [12]: (df.rename_axis("indexname")
```

```
.....:      .rename_axis("columns_name", axis="columns"))
```

```
.....:
```

```
Out[12]:
```

```
columns_name      0      1
```

```
indexname
```

```
0      0.002118  0.405453
```

```
1      0.289092  1.321158
```

```
2     -1.546906 -0.202646
```

```
3     -0.655969  0.193421
```

```
4      0.553439  1.318152
```

The new functionality works well in method chains. Previously these methods only accepted functions or dicts mapping a *label* to a new label. This continues to work as before for function or dict-like values.

### 1.14.1.3 Range Index

A `RangeIndex` has been added to the `Int64Index` sub-classes to support a memory saving alternative for common use cases. This has a similar implementation to the python `range` object (`xrange` in python 2), in that it only stores the start, stop, and step values for the index. It will transparently interact with the user API, converting to `Int64Index` if needed.

This will now be the default constructed index for `NDFrame` objects, rather than previous an `Int64Index`. ([GH939](#), [GH12070](#), [GH12071](#), [GH12109](#), [GH12888](#))

Previous Behavior:

```
In [3]: s = pd.Series(range(1000))
```

```
In [4]: s.index
```

(continues on next page)

(continued from previous page)

```

Out [4]:
Int64Index([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            ...
            990, 991, 992, 993, 994, 995, 996, 997, 998, 999], dtype='int64',
            length=1000)

In [6]: s.index.nbytes
Out [6]: 8000

```

New Behavior:

```

In [13]: s = pd.Series(range(1000))

In [14]: s.index
Out [14]: RangeIndex(start=0, stop=1000, step=1)

In [15]: s.index.nbytes
Out [15]: 80

```

#### 1.14.1.4 Changes to str.extract

The `.str.extract` method takes a regular expression with capture groups, finds the first match in each subject string, and returns the contents of the capture groups ([GH11386](#)).

In v0.18.0, the `expand` argument was added to `extract`.

- `expand=False`: it returns a Series, Index, or DataFrame, depending on the subject and regular expression pattern (same behavior as pre-0.18.0).
- `expand=True`: it always returns a DataFrame, which is more consistent and less confusing from the perspective of a user.

Currently the default is `expand=None` which gives a FutureWarning and uses `expand=False`. To avoid this warning, please explicitly specify `expand`.

```

In [1]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=None)
FutureWarning: currently extract(expand=None) means expand=False (return Index/Series/
↳ DataFrame)
but in a future version of pandas this will be changed to expand=True (return
↳ DataFrame)

Out [1]:
0      1
1      2
2     NaN
dtype: object

```

Extracting a regular expression with one group returns a Series if `expand=False`.

```

In [16]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=False)
Out [16]:
0      1
1      2
2     NaN
dtype: object

```

It returns a DataFrame with one column if `expand=True`.

```
In [17]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=True)
Out[17]:
      0
0     1
1     2
2  NaN
```

Calling on an Index with a regex with exactly one capture group returns an Index if `expand=False`.

```
In [18]: s = pd.Series(["a1", "b2", "c3"], ["A11", "B22", "C33"])

In [19]: s.index
Out[19]: Index(['A11', 'B22', 'C33'], dtype='object')

In [20]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out[20]: Index(['A', 'B', 'C'], dtype='object', name='letter')
```

It returns a DataFrame with one column if `expand=True`.

```
In [21]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
Out[21]:
  letter
0      A
1      B
2      C
```

Calling on an Index with a regex with more than one capture group raises `ValueError` if `expand=False`.

```
>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index
```

It returns a DataFrame if `expand=True`.

```
In [22]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out[22]:
  letter  1
0      A  11
1      B  22
2      C  33
```

In summary, `extract(expand=True)` always returns a DataFrame with a row for every subject string, and a column for every capture group.

#### 1.14.1.5 Addition of `str.extractall`

The `.str.extractall` method was added (GH11386). Unlike `extract`, which returns only the first match.

```
In [23]: s = pd.Series(["a1a2", "b1", "c1"], ["A", "B", "C"])

In [24]: s
Out[24]:
A    a1a2
B      b1
C      c1
dtype: object
```

(continues on next page)



(continued from previous page)

```
In [25]: s.str.extract("(?P<letter>[ab])(?P<digit>\d)", expand=False)
Out[25]:
letter digit
A      a      1
B      b      1
C     NaN     NaN
```

The `extractall` method returns all matches.

```
In [26]: s.str.extractall("(?P<letter>[ab])(?P<digit>\d)")
Out[26]:
letter digit
match
A 0      a      1
  1      a      2
B 0      b      1
```

### 1.14.1.6 Changes to `str.cat`

The method `.str.cat()` concatenates the members of a `Series`. Before, if `NaN` values were present in the `Series`, calling `.str.cat()` on it would return `NaN`, unlike the rest of the `Series.str.*` API. This behavior has been amended to ignore `NaN` values by default. (GH11435).

A new, friendlier `ValueError` is added to protect against the mistake of supplying the `sep` as an arg, rather than as a kwarg. (GH11334).

```
In [27]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ')
Out[27]: 'a b c'

In [28]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(sep=' ', na_rep='?')
Out[28]: 'a b ? c'
```

```
In [2]: pd.Series(['a', 'b', np.nan, 'c']).str.cat(' ')
ValueError: Did you mean to supply a `sep` keyword?
```

### 1.14.1.7 Datetimelike rounding

`DatetimeIndex`, `Timestamp`, `TimedeltaIndex`, `Timedelta` have gained the `.round()`, `.floor()` and `.ceil()` method for datetimelike rounding, flooring and ceiling. (GH4314, GH11963)

Naive datetimes

```
In [29]: dr = pd.date_range('20130101 09:12:56.1234', periods=3)

In [30]: dr
Out[30]:
DatetimeIndex(['2013-01-01 09:12:56.123400', '2013-01-02 09:12:56.123400',
              '2013-01-03 09:12:56.123400'],
              dtype='datetime64[ns]', freq='D')

In [31]: dr.round('s')
Out[31]:
DatetimeIndex(['2013-01-01 09:12:56', '2013-01-02 09:12:56',
              '2013-01-03 09:12:56'],
              dtype='datetime64[ns]', freq='D')
```

(continues on next page)

(continued from previous page)

[illegible]

Tz-aware are rounded, floored and ceiled in local times

```
In [34]: dr = dr.tz_localize('US/Eastern')

In [35]: dr
Out[35]:
DatetimeIndex(['2013-01-01 09:12:56.123400-05:00',
               '2013-01-02 09:12:56.123400-05:00',
               '2013-01-03 09:12:56.123400-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')

In [36]: dr.round('s')
DatetimeIndex(['2013-01-01 09:12:56-05:00', '2013-01-02 09:12:56-05:00',
               '2013-01-03 09:12:56-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq=None)
```

Timedeltas

[illegible]

In addition, `.round()`, `.floor()` and `.ceil()` will be available thru the `.dt` accessor of `Series`.

```
0    2013-01-01 00:00:00-05:00
1    2013-01-02 00:00:00-05:00
2    2013-01-03 00:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

0, 1
1, 2
2, 3

```
Out[47]: Float64Index([0.0, 1.0, 2.0], dtype='float64')
```

(continued from previous page)

```
In [48]: print(s.to_csv(path=None))
```

```
////////////////////////////////////  
↪0,1  
1.0,2  
2.0,3
```

### 1.14.1.9 Changes to dtype assignment behaviors

When a DataFrame's slice is updated with a new slice of the same dtype, the dtype of the DataFrame will now remain the same. (GH10503)

Previous Behavior:

```
In [5]: df = pd.DataFrame({'a': [0, 1, 1],  
                           'b': pd.Series([100, 200, 300], dtype='uint32')})  
  
In [7]: df.dtypes  
Out[7]:  
a      int64  
b      uint32  
dtype: object  
  
In [8]: ix = df['a'] == 1  
  
In [9]: df.loc[ix, 'b'] = df.loc[ix, 'b']  
  
In [11]: df.dtypes  
Out[11]:  
a      int64  
b      int64  
dtype: object
```

New Behavior:

```
In [49]: df = pd.DataFrame({'a': [0, 1, 1],  
.....:                     'b': pd.Series([100, 200, 300], dtype='uint32')})  
.....:  
  
In [50]: df.dtypes  
Out[50]:  
a      int64  
b      uint32  
dtype: object  
  
In [51]: ix = df['a'] == 1  
  
In [52]: df.loc[ix, 'b'] = df.loc[ix, 'b']  
  
In [53]: df.dtypes  
Out[53]:  
a      int64  
b      uint32  
dtype: object
```

When a DataFrame's integer slice is partially updated with a new slice of floats that could potentially be downcasted to integer without losing precision, the dtype of the slice will be set to float instead of integer.

Previous Behavior:

```
In [4]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
                          columns=list('abc'),
                          index=[[4,4,8], [8,10,12]])

In [5]: df
Out[5]:
```

	a	b	c
4 8	1	2	3
10	4	5	6
8 12	7	8	9

```
In [7]: df.ix[4, 'c'] = np.array([0., 1.])

In [8]: df
Out[8]:
```

	a	b	c
4 8	1	2	0
10	4	5	1
8 12	7	8	9

New Behavior:

```
In [54]: df = pd.DataFrame(np.array(range(1,10)).reshape(3,3),
.....:                    columns=list('abc'),
.....:                    index=[[4,4,8], [8,10,12]])
.....:

In [55]: df
Out[55]:
```

	a	b	c
4 8	1	2	3
10	4	5	6
8 12	7	8	9

```
In [56]: df.loc[4, 'c'] = np.array([0., 1.])

In [57]: df
Out[57]:
```

	a	b	c
4 8	1	2	0.0
10	4	5	1.0
8 12	7	8	9.0

### 1.14.1.10 to\_xarray

In a future version of pandas, we will be deprecating Panel and other > 2 ndim objects. In order to provide for continuity, all NDFrame objects have gained the `.to_xarray()` method in order to convert to xarray objects, which has a pandas-like interface for > 2 ndim. (GH11972)

See the [xarray full-documentation](#) here.

```
In [1]: p = Panel(np.arange(2*3*4).reshape(2,3,4))

In [2]: p.to_xarray()
Out[2]:
```

(continues on next page)

(continued from previous page)

```
<xarray.DataArray (items: 2, major_axis: 3, minor_axis: 4)>
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7],
        [ 8,  9, 10, 11]],

       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
Coordinates:
  * items          (items) int64 0 1
  * major_axis     (major_axis) int64 0 1 2
  * minor_axis     (minor_axis) int64 0 1 2 3
```

### 1.14.1.11 Latex Representation

DataFrame has gained a `._repr_latex_()` method in order to allow for conversion to latex in a ipython/jupyter notebook using nbconvert. (GH11778)

Note that this must be activated by setting the option `pd.display.latex.repr=True` (GH12182)

For example, if you have a jupyter notebook you plan to convert to latex using nbconvert, place the statement `pd.display.latex.repr=True` in the first cell to have the contained DataFrame output also stored as latex.

The options `display.latex.escape` and `display.latex.longtable` have also been added to the configuration and are used automatically by the `to_latex` method. See the [available options docs](#) for more info.

### 1.14.1.12 pd.read\_sas() changes

`read_sas` has gained the ability to read SAS7BDAT files, including compressed files. The files can be read in entirety, or incrementally. For full details see [here](#). (GH4052)

### 1.14.1.13 Other enhancements

- Handle truncated floats in SAS xport files (GH11713)
- Added option to hide index in `Series.to_string` (GH11729)
- `read_excel` now supports s3 urls of the format `s3://bucketname/filename` (GH11447)
- add support for `AWS_S3_HOST` env variable when reading from s3 (GH12198)
- A simple version of `Panel.round()` is now implemented (GH11763)
- For Python 3.x, `round(DataFrame)`, `round(Series)`, `round(Panel)` will work (GH11763)
- `sys.getsizeof(obj)` returns the memory usage of a pandas object, including the values it contains (GH11597)
- `Series` gained an `is_unique` attribute (GH11946)
- `DataFrame.quantile` and `Series.quantile` now accept interpolation keyword (GH10174).
- Added `DataFrame.style.format` for more flexible formatting of cell values (GH11692)
- `DataFrame.select_dtypes` now allows the `np.float16` typecode (GH11990)
- `pivot_table()` now accepts most iterables for the `values` parameter (GH12017)

- Added Google BigQuery service account authentication support, which enables authentication on remote servers. (GH11881, GH12572). For further details see [here](#)
- HDFStore is now iterable: `for k in store` is equivalent to `for k in store.keys()` (GH12221).
- Add missing methods/fields to `.dt` for `Period` (GH8848)
- The entire codebase has been PEP-ified (GH12096)

### 1.14.2 Backwards incompatible API changes

- the leading whitespaces have been removed from the output of `.to_string(index=False)` method (GH11833)
- the `out` parameter has been removed from the `Series.round()` method. (GH11763)
- `DataFrame.round()` leaves non-numeric columns unchanged in its return, rather than raises. (GH11885)
- `DataFrame.head(0)` and `DataFrame.tail(0)` return empty frames, rather than `self`. (GH11937)
- `Series.head(0)` and `Series.tail(0)` return empty series, rather than `self`. (GH11937)
- `to_msgpack` and `read_msgpack` encoding now defaults to `'utf-8'`. (GH12170)
- the order of keyword arguments to text file parsing functions (`.read_csv()`, `.read_table()`, `.read_fwf()`) changed to group related arguments. (GH11555)
- `NaTType.isoformat` now returns the string `'NaT'` to allow the result to be passed to the constructor of `Timestamp`. (GH12300)

#### 1.14.2.1 NaT and Timedelta operations

`NaT` and `Timedelta` have expanded arithmetic operations, which are extended to `Series` arithmetic where applicable. Operations defined for `datetime64[ns]` or `timedelta64[ns]` are now also defined for `NaT` (GH11564).

`NaT` now supports arithmetic operations with integers and floats.

```
In [58]: pd.NaT * 1
Out[58]: NaT

In [59]: pd.NaT * 1.5
\\Out[59]: NaT

In [60]: pd.NaT / 2
\\Out[60]: NaT

In [61]: pd.NaT * np.nan
\\Out[61]: NaT
```

`NaT` defines more arithmetic operations with `datetime64[ns]` and `timedelta64[ns]`.

```
In [62]: pd.NaT / pd.NaT
Out[62]: nan

In [63]: pd.Timedelta('1s') / pd.NaT
\\Out[63]: nan
```

`NaT` may represent either a `datetime64[ns]` null or a `timedelta64[ns]` null. Given the ambiguity, it is treated as a `timedelta64[ns]`, which allows more operations to succeed.

```
In [64]: pd.NaT + pd.NaT
Out[64]: NaT

# same as
In [65]: pd.Timedelta('1s') + pd.Timedelta('1s')
Out[65]: Timedelta('0 days 00:00:02')
```

as opposed to

```
In [3]: pd.Timestamp('19900315') + pd.Timestamp('19900315')
TypeError: unsupported operand type(s) for +: 'Timestamp' and 'Timestamp'
```

However, when wrapped in a Series whose dtype is `datetime64[ns]` or `timedelta64[ns]`, the dtype information is respected.

```
In [1]: pd.Series([pd.NaT], dtype='<M8[ns]') + pd.Series([pd.NaT], dtype='<M8[ns]')
TypeError: can only operate on a datetimes for subtraction,
but the operator [__add__] was passed
```

```
In [66]: pd.Series([pd.NaT], dtype='<m8[ns]') + pd.Series([pd.NaT], dtype='<m8[ns]')
Out[66]:
0      NaT
dtype: timedelta64[ns]
```

Timedelta division by floats now works.

```
In [67]: pd.Timedelta('1s') / 2.0
Out[67]: Timedelta('0 days 00:00:00.500000')
```

Subtraction by Timedelta in a Series by a Timestamp works (GH11925)

```
In [68]: ser = pd.Series(pd.timedelta_range('1 day', periods=3))

In [69]: ser
Out[69]:
0    1 days
1    2 days
2    3 days
dtype: timedelta64[ns]

In [70]: pd.Timestamp('2012-01-01') - ser
Out[70]:
0    2011-12-31
1    2011-12-30
2    2011-12-29
dtype: datetime64[ns]
```

`NaT.isoformat()` now returns `'NaT'`. This change allows `pd.Timestamp` to rehydrate any timestamp like object from its `isoformat` (GH12300).

### 1.14.2.2 Changes to msgpack

Forward incompatible changes in msgpack writing format were made over 0.17.0 and 0.18.0; older versions of pandas cannot read files packed by newer versions (GH12129, GH10527)

Bugs in `to_msgpack` and `read_msgpack` introduced in 0.17.0 and fixed in 0.18.0, caused files packed in Python 2 unreadable by Python 3 (GH12142). The following table describes the backward and forward compat of msgpacks.



`Series.rank` and `DataFrame.rank` now have the same signature ([GH11759](#))

Previous signature

```
In [3]: pd.Series([0,1]).rank(method='average', na_option='keep',
                                ascending=True, pct=False)

Out[3]:
0    1
1    2
dtype: float64

In [4]: pd.DataFrame([0,1]).rank(axis=0, numeric_only=None,
                                   method='average', na_option='keep',
                                   ascending=True, pct=False)

Out[4]:
0
0  1
1  2
```

[illegible]

### 1.14.2.4 Bug in QuarterBegin with n=0

In previous versions, the behavior of the QuarterBegin offset was inconsistent depending on the date when the `n` parameter was 0. (GH11406)

The general semantics of anchored offsets for `n=0` is to not move the date when it is an anchor point (e.g., a quarter start date), and otherwise roll forward to the next anchor point.

```
In [73]: d = pd.Timestamp('2014-02-01')

In [74]: d
Out[74]: Timestamp('2014-02-01 00:00:00')

In [75]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[75]: Timestamp('2014-02-01 00:00:00')

In [76]: d + pd.offsets.QuarterBegin(n=0, startingMonth=1)
Out[76]: Timestamp('2014-04-01 00:00:00')
```

For the QuarterBegin offset in previous versions, the date would be rolled *backwards* if date was in the same month as the quarter start date.

```
In [3]: d = pd.Timestamp('2014-02-15')

In [4]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[4]: Timestamp('2014-02-01 00:00:00')
```

This behavior has been corrected in version 0.18.0, which is consistent with other anchored offsets like MonthBegin and YearBegin.

```
In [77]: d = pd.Timestamp('2014-02-15')

In [78]: d + pd.offsets.QuarterBegin(n=0, startingMonth=2)
Out[78]: Timestamp('2014-05-01 00:00:00')
```

### 1.14.2.5 Resample API

Like the change in the window functions API [above](#), `.resample(...)` is changing to have a more groupby-like API. (GH11732, GH12702, GH12202, GH12332, GH12334, GH12348, GH12448).

```
In [79]: np.random.seed(1234)

In [80]: df = pd.DataFrame(np.random.rand(10,4),
.....:                    columns=list('ABCD'),
.....:                    index=pd.date_range('2010-01-01 09:00:00', periods=10,
.....:                    freq='s'))

In [81]: df
Out[81]:
```

	A	B	C	D
2010-01-01 09:00:00	0.191519	0.622109	0.437728	0.785359
2010-01-01 09:00:01	0.779976	0.272593	0.276464	0.801872
2010-01-01 09:00:02	0.958139	0.875933	0.357817	0.500995
2010-01-01 09:00:03	0.683463	0.712702	0.370251	0.561196

(continues on next page)

(continued from previous page)

2010-01-01 09:00:04	0.503083	0.013768	0.772827	0.882641
2010-01-01 09:00:05	0.364886	0.615396	0.075381	0.368824
2010-01-01 09:00:06	0.933140	0.651378	0.397203	0.788730
2010-01-01 09:00:07	0.316836	0.568099	0.869127	0.436173
2010-01-01 09:00:08	0.802148	0.143767	0.704261	0.704581
2010-01-01 09:00:09	0.218792	0.924868	0.442141	0.909316

**Previous API:**

You would write a resampling operation that immediately evaluates. If a how parameter was not provided, it would default to how='mean'.

```
In [6]: df.resample('2s')
```

```
Out [6]:
```

	A	B	C	D
2010-01-01 09:00:00	0.485748	0.447351	0.357096	0.793615
2010-01-01 09:00:02	0.820801	0.794317	0.364034	0.531096
2010-01-01 09:00:04	0.433985	0.314582	0.424104	0.625733
2010-01-01 09:00:06	0.624988	0.609738	0.633165	0.612452
2010-01-01 09:00:08	0.510470	0.534317	0.573201	0.806949

You could also specify a how directly

```
In [7]: df.resample('2s', how='sum')
```

```
Out [7]:
```

	A	B	C	D
2010-01-01 09:00:00	0.971495	0.894701	0.714192	1.587231
2010-01-01 09:00:02	1.641602	1.588635	0.728068	1.062191
2010-01-01 09:00:04	0.867969	0.629165	0.848208	1.251465
2010-01-01 09:00:06	1.249976	1.219477	1.266330	1.224904
2010-01-01 09:00:08	1.020940	1.068634	1.146402	1.613897

**New API:**

Now, you can write `.resample(...)` as a 2-stage operation like `.groupby(...)`, which yields a Resampler.

```
In [82]: r = df.resample('2s')
```

```
In [83]: r
```

```
Out [83]: DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left, label=left, ↵
↪convention=start, base=0]
```

**Downsampling**

You can then use this object to perform operations. These are downsampling operations (going from a higher frequency to a lower one).

```
In [84]: r.mean()
```

```
Out [84]:
```

	A	B	C	D
2010-01-01 09:00:00	0.485748	0.447351	0.357096	0.793615
2010-01-01 09:00:02	0.820801	0.794317	0.364034	0.531096
2010-01-01 09:00:04	0.433985	0.314582	0.424104	0.625733
2010-01-01 09:00:06	0.624988	0.609738	0.633165	0.612452
2010-01-01 09:00:08	0.510470	0.534317	0.573201	0.806949

```
In [85]: r.sum()
```

```
Out [85]:
```

	A	B	C	D
2010-01-01 09:00:00	0.971495	0.894701	0.714192	1.587231
2010-01-01 09:00:02	1.641602	1.588635	0.728068	1.062191
2010-01-01 09:00:04	0.867969	0.629165	0.848208	1.251465
2010-01-01 09:00:06	1.249976	1.219477	1.266330	1.224904
2010-01-01 09:00:08	1.020940	1.068634	1.146402	1.613897

Furthermore, `resample` now supports `getitem` operations to perform the resample on specific columns.

```
In [86]: r[['A', 'C']].mean()
```

```
Out [86]:
```

	A	C
2010-01-01 09:00:00	0.485748	0.357096
2010-01-01 09:00:02	0.820801	0.364034
2010-01-01 09:00:04	0.433985	0.424104
2010-01-01 09:00:06	0.624988	0.633165
2010-01-01 09:00:08	0.510470	0.573201

and `.aggregate` type operations.

```
In [87]: r.agg({'A' : 'mean', 'B' : 'sum'})
```

```
Out [87]:
```

	A	B
2010-01-01 09:00:00	0.485748	0.894701
2010-01-01 09:00:02	0.820801	1.588635
2010-01-01 09:00:04	0.433985	0.629165
2010-01-01 09:00:06	0.624988	1.219477
2010-01-01 09:00:08	0.510470	1.068634

These accessors can of course, be combined

```
In [88]: r[['A', 'B']].agg(['mean', 'sum'])
```

```
Out [88]:
```

	A		B	
	mean	sum	mean	sum
2010-01-01 09:00:00	0.485748	0.971495	0.447351	0.894701
2010-01-01 09:00:02	0.820801	1.641602	0.794317	1.588635
2010-01-01 09:00:04	0.433985	0.867969	0.314582	0.629165
2010-01-01 09:00:06	0.624988	1.249976	0.609738	1.219477
2010-01-01 09:00:08	0.510470	1.020940	0.534317	1.068634

## Upsampling

Upsampling operations take you from a lower frequency to a higher frequency. These are now performed with the `Resampler` objects with `backfill()`, `ffill()`, `fillna()` and `asfreq()` methods.

```
In [89]: s = pd.Series(np.arange(5, dtype='int64'),
.....:                 index=date_range('2010-01-01', periods=5, freq='Q'))
.....:
```

```
In [90]: s
```

```
Out [90]:
2010-03-31    0
```

(continues on next page)

(continued from previous page)

```

2010-06-30    1
2010-09-30    2
2010-12-31    3
2011-03-31    4
Freq: Q-DEC, dtype: int64

```

### Previously

```

In [6]: s.resample('M', fill_method='ffill')
Out [6]:
2010-03-31    0
2010-04-30    0
2010-05-31    0
2010-06-30    1
2010-07-31    1
2010-08-31    1
2010-09-30    2
2010-10-31    2
2010-11-30    2
2010-12-31    3
2011-01-31    3
2011-02-28    3
2011-03-31    4
Freq: M, dtype: int64

```

### New API

```

In [91]: s.resample('M').ffill()
Out [91]:
2010-03-31    0
2010-04-30    0
2010-05-31    0
2010-06-30    1
2010-07-31    1
2010-08-31    1
2010-09-30    2
2010-10-31    2
2010-11-30    2
2010-12-31    3
2011-01-31    3
2011-02-28    3
2011-03-31    4
Freq: M, dtype: int64

```

**Note:** In the new API, you can either downsample OR upsample. The prior implementation would allow you to pass an aggregator function (like `mean`) even though you were upsampling, providing a bit of confusion.

### Previous API will work but with deprecations

**Warning:** This new API for `resample` includes some internal changes for the prior-to-0.18.0 API, to work with a deprecation warning in most cases, as the `resample` operation returns a deferred object. We can intercept operations

and just do what the (pre 0.18.0) API did (with a warning). Here is a typical use case:

```
In [4]: r = df.resample('2s')
```

```
In [6]: r*10
```

```
pandas/tseries/resample.py:80: FutureWarning: .resample() is now a deferred_
→operation
use .resample(...).mean() instead of .resample(...)
```

```
Out [6]:
```

	A	B	C	D
2010-01-01 09:00:00	4.857476	4.473507	3.570960	7.936154
2010-01-01 09:00:02	8.208011	7.943173	3.640340	5.310957
2010-01-01 09:00:04	4.339846	3.145823	4.241039	6.257326
2010-01-01 09:00:06	6.249881	6.097384	6.331650	6.124518
2010-01-01 09:00:08	5.104699	5.343172	5.732009	8.069486

However, getting and assignment operations directly on a Resampler will raise a ValueError:

```
In [7]: r.iloc[0] = 5
```

```
ValueError: .resample() is now a deferred operation
use .resample(...).mean() instead of .resample(...)
```

There is a situation where the new API can not perform all the operations when using original code. This code is intending to resample every 2s, take the mean AND then take the min of those results.

```
In [4]: df.resample('2s').min()
```

```
Out [4]:
```

```
A    0.433985
B    0.314582
C    0.357096
D    0.531096
dtype: float64
```

The new API will:

```
In [92]: df.resample('2s').min()
```

```
Out [92]:
```

	A	B	C	D
2010-01-01 09:00:00	0.191519	0.272593	0.276464	0.785359
2010-01-01 09:00:02	0.683463	0.712702	0.357817	0.500995
2010-01-01 09:00:04	0.364886	0.013768	0.075381	0.368824
2010-01-01 09:00:06	0.316836	0.568099	0.397203	0.436173
2010-01-01 09:00:08	0.218792	0.143767	0.442141	0.704581

The good news is the return dimensions will differ between the new API and the old API, so this should loudly raise an exception.

To replicate the original operation

```
In [93]: df.resample('2s').mean().min()
```

```
Out [93]:
```

```
A    0.433985
B    0.314582
C    0.357096
D    0.531096
dtype: float64
```

### 1.14.2.6 Changes to eval

In prior versions, new columns assignments in an `eval` expression resulted in an inplace change to the `DataFrame`. (GH9297, GH8664, GH10486)

```
In [94]: df = pd.DataFrame({'a': np.linspace(0, 10, 5), 'b': range(5)})
```

```
In [95]: df
```

```
Out[95]:
```

	a	b
0	0.0	0
1	2.5	1
2	5.0	2
3	7.5	3
4	10.0	4

```
In [12]: df.eval('c = a + b')
```

FutureWarning: eval expressions containing an assignment currently default to `inplace=True`. This will change in a future version of pandas, use `inplace=False` to avoid this warning.

```
In [13]: df
```

```
Out[13]:
```

	a	b	c
0	0.0	0	0.0
1	2.5	1	3.5
2	5.0	2	7.0
3	7.5	3	10.5
4	10.0	4	14.0

In version 0.18.0, a new `inplace` keyword was added to choose whether the assignment should be done inplace or return a copy.

```
In [96]: df
```

```
Out[96]:
```

	a	b	c
0	0.0	0	0.0
1	2.5	1	3.5
2	5.0	2	7.0
3	7.5	3	10.5
4	10.0	4	14.0

```
In [97]: df.eval('d = c - b', inplace=False)
```

```
Out[97]:
```

	a	b	c	d
0	0.0	0	0.0	0.0
1	2.5	1	3.5	2.5
2	5.0	2	7.0	5.0
3	7.5	3	10.5	7.5
4	10.0	4	14.0	10.0

```
In [98]: df
```

```
Out[98]:
```

	a	b	c
0	0.0	0	0.0
1	2.5	1	3.5
2	5.0	2	7.0
3	7.5	3	10.5
4	10.0	4	14.0

(continues on next page)

(continued from previous page)

```

0    0.0  0    0.0
1    2.5  1    3.5
2    5.0  2    7.0
3    7.5  3   10.5
4   10.0  4   14.0

```

```
In [99]: df.eval('d = c - b', inplace=True)
```

```
In [100]: df
```

```
Out[100]:
```

	a	b	c	d
0	0.0	0	0.0	0.0
1	2.5	1	3.5	2.5
2	5.0	2	7.0	5.0
3	7.5	3	10.5	7.5
4	10.0	4	14.0	10.0

**Warning:** For backwards compatibility, `inplace` defaults to `True` if not specified. This will change in a future version of pandas. If your code depends on an inplace assignment you should update to explicitly set `inplace=True`

The `inplace` keyword parameter was also added the `query` method.

```
In [101]: df.query('a > 5')
```

```
Out[101]:
```

	a	b	c	d
3	7.5	3	10.5	7.5
4	10.0	4	14.0	10.0

```
In [102]: df.query('a > 5', inplace=True)
```

```
In [103]: df
```

```
Out[103]:
```

	a	b	c	d
3	7.5	3	10.5	7.5
4	10.0	4	14.0	10.0

**Warning:** Note that the default value for `inplace` in a `query` is `False`, which is consistent with prior versions.

`eval` has also been updated to allow multi-line expressions for multiple assignments. These expressions will be evaluated one at a time in order. Only assignments are valid for multi-line expressions.

```
In [104]: df
```

```
Out[104]:
```

	a	b	c	d
3	7.5	3	10.5	7.5
4	10.0	4	14.0	10.0

```
In [105]: df.eval("""
.....: e = d + a
.....: f = e - 22
.....: g = f / 2.0""", inplace=True)
```

(continues on next page)



(continued from previous page)

```
.....:
In [106]: df
Out[106]:
```

	a	b	c	d	e	f	g
3	7.5	3	10.5	7.5	15.0	-7.0	-3.5
4	10.0	4	14.0	10.0	20.0	-2.0	-1.0

### 1.14.2.7 Other API Changes

- `DataFrame.between_time` and `Series.between_time` now only parse a fixed set of time strings. Parsing of date strings is no longer supported and raises a `ValueError`. (GH11818)

```
In [107]: s = pd.Series(range(10), pd.date_range('2015-01-01', freq='H',
↳ periods=10))

In [108]: s.between_time("7:00am", "9:00am")
Out[108]:
2015-01-01 07:00:00    7
2015-01-01 08:00:00    8
2015-01-01 09:00:00    9
Freq: H, dtype: int64
```

This will now raise.

```
In [2]: s.between_time('20150101 07:00:00', '20150101 09:00:00')
ValueError: Cannot convert arg ['20150101 07:00:00'] to a time.
```

- `.memory_usage()` now includes values in the index, as does `memory_usage` in `.info()` (GH11597)
- `DataFrame.to_latex()` now supports non-ascii encodings (eg `utf-8`) in Python 2 with the parameter `encoding` (GH7061)
- `pandas.merge()` and `DataFrame.merge()` will show a specific error message when trying to merge with an object that is not of type `DataFrame` or a subclass (GH12081)
- `DataFrame.unstack` and `Series.unstack` now take `fill_value` keyword to allow direct replacement of missing values when an unstack results in missing values in the resulting `DataFrame`. As an added benefit, specifying `fill_value` will preserve the data type of the original stacked data. (GH9746)
- As part of the new API for *window functions* and *resampling*, aggregation functions have been clarified, raising more informative error messages on invalid aggregations. (GH9052). A full set of examples are presented in *groupby*.
- Statistical functions for `NDFrame` objects (like `sum()`, `mean()`, `min()`) will now raise if non-numpy-compatible arguments are passed in for `**kwargs` (GH12301)
- `.to_latex` and `.to_html` gain a `decimal` parameter like `.to_csv`; the default is `'.'` (GH12031)
- More helpful error message when constructing a `DataFrame` with empty data but with indices (GH8020)
- `.describe()` will now properly handle `bool` dtype as a categorical (GH6625)
- More helpful error message with an invalid `.transform` with user defined input (GH10165)
- Exponentially weighted functions now allow specifying `alpha` directly (GH10789) and raise `ValueError` if parameters violate  $0 < \alpha \leq 1$  (GH12492)

### 1.14.2.8 Deprecations

- The functions `pd.rolling_*`, `pd.expanding_*`, and `pd.ewm*` are deprecated and replaced by the corresponding method call. Note that the new suggested syntax includes all of the arguments (even if default) ([GH11603](#))

```
In [1]: s = pd.Series(range(3))

In [2]: pd.rolling_mean(s, window=2, min_periods=1)
FutureWarning: pd.rolling_mean is deprecated for Series and
will be removed in a future version, replace with
Series.rolling(min_periods=1, window=2, center=False).mean()

Out [2]:
0    0.0
1    0.5
2    1.5
dtype: float64

In [3]: pd.rolling_cov(s, s, window=2)
FutureWarning: pd.rolling_cov is deprecated for Series and
will be removed in a future version, replace with
Series.rolling(window=2).cov(other=<Series>)

Out [3]:
0    NaN
1    0.5
2    0.5
dtype: float64
```

- The `freq` and `how` arguments to the `.rolling`, `.expanding`, and `.ewm` (new) functions are deprecated, and will be removed in a future version. You can simply resample the input prior to creating a window function. ([GH11603](#)).

For example, instead of `s.rolling(window=5, freq='D').max()` to get the max value on a rolling 5 Day window, one could use `s.resample('D').mean().rolling(window=5).max()`, which first resamples the data to daily data, then provides a rolling 5 day window.

- `pd.tseries.frequencies.get_offset_name` function is deprecated. Use `offset's .freqstr` property as alternative ([GH11192](#))
- `pandas.stats.fama_macbeth` routines are deprecated and will be removed in a future version ([GH6077](#))
- `pandas.stats.ols`, `pandas.stats.plm` and `pandas.stats.var` routines are deprecated and will be removed in a future version ([GH6077](#))
- show a `FutureWarning` rather than a `DeprecationWarning` on using long-time deprecated syntax in `HDFStore.select`, where the `where` clause is not a string-like ([GH12027](#))
- The `pandas.options.display.mpl_style` configuration has been deprecated and will be removed in a future version of pandas. This functionality is better handled by matplotlib's [style sheets](#) ([GH11783](#)).

### 1.14.2.9 Removal of deprecated float indexers

In [GH4892](#) indexing with floating point numbers on a non-`Float64Index` was deprecated (in version 0.14.0). In 0.18.0, this deprecation warning is removed and these will now raise a `TypeError`. ([GH12165](#), [GH12333](#))

```
In [109]: s = pd.Series([1, 2, 3], index=[4, 5, 6])

In [110]: s
```

(continues on next page)

(continued from previous page)

```

Out[110]:
4      1
5      2
6      3
dtype: int64

In [111]: s2 = pd.Series([1, 2, 3], index=list('abc'))

In [112]: s2
Out[112]:
a      1
b      2
c      3
dtype: int64

```

**Previous Behavior:**

```

# this is label indexing
In [2]: s[5.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not
↳floating point
Out[2]: 2

# this is positional indexing
In [3]: s.iloc[1.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not
↳floating point
Out[3]: 2

# this is label indexing
In [4]: s.loc[5.0]
FutureWarning: scalar indexers for index type Int64Index should be integers and not
↳floating point
Out[4]: 2

# .ix would coerce 1.0 to the positional 1, and index
In [5]: s2.ix[1.0] = 10
FutureWarning: scalar indexers for index type Index should be integers and not
↳floating point

In [6]: s2
Out[6]:
a      1
b     10
c      3
dtype: int64

```

**New Behavior:**

For iloc, getting & setting via a float scalar will always raise.

```

In [3]: s.iloc[2.0]
TypeError: cannot do label indexing on <class 'pandas.indexes.numeric.Int64Index'>
↳with these indexers [2.0] of <type 'float'>

```

Other indexers will coerce to a like integer for both getting and setting. The FutureWarning has been dropped for .loc, .ix and [].

```
In [113]: s[5.0]
Out[113]: 2

In [114]: s.loc[5.0]
\\\\\\\\\\\\\\\\\\\\Out[114]: 2
```

and setting

```
In [115]: s_copy = s.copy()

In [116]: s_copy[5.0] = 10

In [117]: s_copy
Out[117]:
4      1
5     10
6      3
dtype: int64

In [118]: s_copy = s.copy()

In [119]: s_copy.loc[5.0] = 10

In [120]: s_copy
Out[120]:
4      1
5     10
6      3
dtype: int64
```

Positional setting with `.ix` and a float indexer will ADD this value to the index, rather than previously setting the value by position.

```
In [3]: s2.ix[1.0] = 10
In [4]: s2
Out[4]:
a      1
b      2
c      3
1.0    10
dtype: int64
```

Slicing will also coerce integer-like floats to integers for a non-Float64Index.

```
In [121]: s.loc[5.0:6]
Out[121]:
5      2
6      3
dtype: int64
```

Note that for floats that are NOT coercible to ints, the label based bounds will be excluded

```
In [122]: s.loc[5.1:6]
Out[122]:
6      3
dtype: int64
```

Float indexing on a Float64Index is unchanged.

```
In [123]: s = pd.Series([1, 2, 3], index=np.arange(3.))
In [124]: s[1.0]
Out[124]: 2
In [125]: s[1.0:2.5]
\\\\\\\\\\\\\\\\\\\\Out[125]:
1.0    2
2.0    3
dtype: int64
```

#### 1.14.2.10 Removal of prior version deprecations/changes

- Removal of `rolling_corr_pairwise` in favor of `.rolling().corr(pairwise=True)` ([GH4950](#))
- Removal of `expanding_corr_pairwise` in favor of `.expanding().corr(pairwise=True)` ([GH4950](#))
- Removal of `DataMatrix` module. This was not imported into the pandas namespace in any event ([GH12111](#))
- Removal of `cols` keyword in favor of `subset` in `DataFrame.duplicated()` and `DataFrame.drop_duplicates()` ([GH6680](#))
- Removal of the `read_frame` and `frame_query` (both aliases for `pd.read_sql`) and `write_frame` (alias of `to_sql`) functions in the `pd.io.sql` namespace, deprecated since 0.14.0 ([GH6292](#)).
- Removal of the `order` keyword from `.factorize()` ([GH6930](#))

#### 1.14.3 Performance Improvements

- Improved performance of `andrews_curves` ([GH11534](#))
- Improved huge `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex`'s ops performance including `NaT` ([GH10277](#))
- Improved performance of `pandas.concat` ([GH11958](#))
- Improved performance of `StataReader` ([GH11591](#))
- Improved performance in construction of `Categoricals` with `Series` of datetimes containing `NaT` ([GH12077](#))
- Improved performance of ISO 8601 date parsing for dates without separators ([GH11899](#)), leading zeros ([GH11871](#)) and with whitespace preceding the time zone ([GH9714](#))

#### 1.14.4 Bug Fixes

- Bug in `GroupBy.size` when data-frame is empty. ([GH11699](#))
- Bug in `Period.end_time` when a multiple of time period is requested ([GH11738](#))
- Regression in `.clip` with tz-aware datetimes ([GH11838](#))
- Bug in `date_range` when the boundaries fell on the frequency ([GH11804](#), [GH12409](#))
- Bug in consistency of passing nested dicts to `.groupby(...).agg(...)` ([GH9052](#))
- Accept unicode in `Timedelta` constructor ([GH11995](#))

- Bug in value label reading for `StataReader` when reading incrementally ([GH12014](#))
- Bug in vectorized `DateOffset` when `n` parameter is 0 ([GH11370](#))
- Compat for numpy 1.11 w.r.t. `NaT` comparison changes ([GH12049](#))
- Bug in `read_csv` when reading from a `StringIO` in threads ([GH11790](#))
- Bug in not treating `NaT` as a missing value in datetimelikes when factorizing & with `Categoricals` ([GH12077](#))
- Bug in `getitem` when the values of a `Series` were tz-aware ([GH12089](#))
- Bug in `Series.str.get_dummies` when one of the variables was 'name' ([GH12180](#))
- Bug in `pd.concat` while concatenating tz-aware `NaT` series. ([GH11693](#), [GH11755](#), [GH12217](#))
- Bug in `pd.read_stata` with version `<= 108` files ([GH12232](#))
- Bug in `Series.resample` using a frequency of `Nano` when the index is a `DatetimeIndex` and contains non-zero nanosecond parts ([GH12037](#))
- Bug in resampling with `.nunique` and a sparse index ([GH12352](#))
- Removed some compiler warnings ([GH12471](#))
- Work around compat issues with `boto` in python 3.5 ([GH11915](#))
- Bug in `NaT` subtraction from `Timestamp` or `DatetimeIndex` with timezones ([GH11718](#))
- Bug in subtraction of `Series` of a single tz-aware `Timestamp` ([GH12290](#))
- Use compat iterators in PY2 to support `.next()` ([GH12299](#))
- Bug in `Timedelta.round` with negative values ([GH11690](#))
- Bug in `.loc` against `CategoricalIndex` may result in normal `Index` ([GH11586](#))
- Bug in `DataFrame.info` when duplicated column names exist ([GH11761](#))
- Bug in `.copy` of datetime tz-aware objects ([GH11794](#))
- Bug in `Series.apply` and `Series.map` where `timedelta64` was not boxed ([GH11349](#))
- Bug in `DataFrame.set_index()` with tz-aware `Series` ([GH12358](#))
- Bug in subclasses of `DataFrame` where `AttributeError` did not propagate ([GH11808](#))
- Bug groupby on tz-aware data where selection not returning `Timestamp` ([GH11616](#))
- Bug in `pd.read_clipboard` and `pd.to_clipboard` functions not supporting Unicode; upgrade included `pyperclip` to v1.5.15 ([GH9263](#))
- Bug in `DataFrame.query` containing an assignment ([GH8664](#))
- Bug in `from_msgpack` where `__contains__()` fails for columns of the unpacked `DataFrame`, if the `DataFrame` has object columns. ([GH11880](#))
- Bug in `.resample` on categorical data with `TimedeltaIndex` ([GH12169](#))
- Bug in timezone info lost when broadcasting scalar datetime to `DataFrame` ([GH11682](#))
- Bug in `Index` creation from `Timestamp` with mixed tz coerces to UTC ([GH11488](#))
- Bug in `to_numeric` where it does not raise if input is more than one dimension ([GH11776](#))
- Bug in parsing timezone offset strings with non-zero minutes ([GH11708](#))
- Bug in `df.plot` using incorrect colors for bar plots under `matplotlib` 1.5+ ([GH11614](#))

- Bug in the `groupby` `plot` method when using keyword arguments (GH11805).
- Bug in `DataFrame.duplicated` and `drop_duplicates` causing spurious matches when setting `keep=False` (GH11864)
- Bug in `.loc` result with duplicated key may have `Index` with incorrect `dtype` (GH11497)
- Bug in `pd.rolling_median` where memory allocation failed even with sufficient memory (GH11696)
- Bug in `DataFrame.style` with spurious zeros (GH12134)
- Bug in `DataFrame.style` with integer columns not starting at 0 (GH12125)
- Bug in `.style.bar` may not rendered properly using specific browser (GH11678)
- Bug in rich comparison of `Timedelta` with a `numpy.array` of `Timedelta` that caused an infinite recursion (GH11835)
- Bug in `DataFrame.round` dropping column index name (GH11986)
- Bug in `df.replace` while replacing value in mixed dtype `Dataframe` (GH11698)
- Bug in `Index` prevents copying name of passed `Index`, when a new name is not provided (GH11193)
- Bug in `read_excel` failing to read any non-empty sheets when empty sheets exist and `sheetname=None` (GH11711)
- Bug in `read_excel` failing to raise `NotImplemented` error when keywords `parse_dates` and `date_parser` are provided (GH11544)
- Bug in `read_sql` with `pymysql` connections failing to return chunked data (GH11522)
- Bug in `.to_csv` ignoring formatting parameters `decimal`, `na_rep`, `float_format` for float indexes (GH11553)
- Bug in `Int64Index` and `Float64Index` preventing the use of the modulo operator (GH9244)
- Bug in `MultiIndex.drop` for not lexsorted multi-indexes (GH12078)
- Bug in `DataFrame` when masking an empty `DataFrame` (GH11859)
- Bug in `.plot` potentially modifying the `colors` input when the number of columns didn't match the number of series provided (GH12039).
- Bug in `Series.plot` failing when index has a `CustomBusinessDay` frequency (GH7222).
- Bug in `.to_sql` for `datetime.time` values with `sqlite` fallback (GH8341)
- Bug in `read_excel` failing to read data with one column when `squeeze=True` (GH12157)
- Bug in `read_excel` failing to read one empty column (GH12292, GH9002)
- Bug in `.groupby` where a `KeyError` was not raised for a wrong column if there was only one row in the dataframe (GH11741)
- Bug in `.read_csv` with `dtype` specified on empty data producing an error (GH12048)
- Bug in `.read_csv` where strings like `'2E'` are treated as valid floats (GH12237)
- Bug in building `pandas` with debugging symbols (GH12123)
- Removed `millisecond` property of `DatetimeIndex`. This would always raise a `ValueError` (GH12019).
- Bug in `Series` constructor with read-only data (GH11502)
- Removed `pandas.util.testing.choice()`. Should use `np.random.choice()`, instead. (GH12386)

- Bug in `.loc` setitem indexer preventing the use of a TZ-aware `DatetimeIndex` ([GH12050](#))
- Bug in `.style` indexes and multi-indexes not appearing ([GH11655](#))
- Bug in `to_msgpack` and `from_msgpack` which did not correctly serialize or deserialize `NaT` ([GH12307](#)).
- Bug in `.skew` and `.kurt` due to roundoff error for highly similar values ([GH11974](#))
- Bug in `Timestamp` constructor where microsecond resolution was lost if `HHMMSS` were not separated with `‘.’` ([GH10041](#))
- Bug in `buffer_rd_bytes` `src->buffer` could be freed more than once if reading failed, causing a segfault ([GH12098](#))
- Bug in `crosstab` where arguments with non-overlapping indexes would return a `KeyError` ([GH10291](#))
- Bug in `DataFrame.apply` in which reduction was not being prevented for cases in which `dtype` was not a numpy dtype ([GH12244](#))
- Bug when initializing categorical series with a scalar value. ([GH12336](#))
- Bug when specifying a UTC `DatetimeIndex` by setting `utc=True` in `.to_datetime` ([GH11934](#))
- Bug when increasing the buffer size of CSV reader in `read_csv` ([GH12494](#))
- Bug when setting columns of a `DataFrame` with duplicate column names ([GH12344](#))

## 1.15 v0.17.1 (November 21, 2015)

---

**Note:** We are proud to announce that *pandas* has become a sponsored project of the ([NumFOCUS organization](#)). This will help ensure the success of development of *pandas* as a world-class open-source project.

---

This is a minor bug-fix release from 0.17.0 and includes a large number of bug fixes along several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Support for Conditional HTML Formatting, see [here](#)
- Releasing the GIL on the csv reader & other ops, see [here](#)
- Fixed regression in `DataFrame.drop_duplicates` from 0.16.2, causing incorrect results on integer values ([GH11376](#))

### What's new in v0.17.1

- *New features*
  - *Conditional HTML Formatting*
- *Enhancements*
- *API changes*
  - *Deprecations*
- *Performance Improvements*
- *Bug Fixes*



## 1.15.1 New features

### 1.15.1.1 Conditional HTML Formatting

**Warning:** This is a new feature and is under active development. We'll be adding features and possibly making breaking changes in future releases. Feedback is [welcome](#).

We've added *experimental* support for conditional HTML formatting: the visual styling of a DataFrame based on the data. The styling is accomplished with HTML and CSS. Accesses the styler class with the `pandas.DataFrame.style` attribute, an instance of `Styler` with your data attached.

Here's a quick example:

```
In [1]: np.random.seed(123)

In [2]: df = DataFrame(np.random.randn(10, 5), columns=list('abcde'))

In [3]: html = df.style.background_gradient(cmap='viridis', low=.5)
```

We can render the HTML to get the following table.

`Styler` interacts nicely with the Jupyter Notebook. See the [documentation](#) for more.

## 1.15.2 Enhancements

- `DatetimeIndex` now supports conversion to strings with `astype(str)` (GH10442)
- Support for compression (gzip/bz2) in `pandas.DataFrame.to_csv()` (GH7615)
- `pd.read_*` functions can now also accept `pathlib.Path`, or `py._path.local.LocalPath` objects for the `filepath_or_buffer` argument. (GH11033) - The `DataFrame` and `Series` functions `.to_csv()`, `.to_html()` and `.to_latex()` can now handle paths beginning with tildes (e.g. `~/Documents/`) (GH11438)
- `DataFrame` now uses the fields of a `namedtuple` as columns, if columns are not supplied (GH11181)
- `DataFrame.itertuples()` now returns `namedtuple` objects, when possible. (GH11269, GH11625)
- Added `axvlines_kwds` to parallel coordinates plot (GH10709)
- Option to `.info()` and `.memory_usage()` to provide for deep introspection of memory consumption. Note that this can be expensive to compute and therefore is an optional parameter. (GH11595)

```
In [4]: df = DataFrame({'A' : ['foo']*1000})

In [5]: df['B'] = df['A'].astype('category')

# shows the '+' as we have object dtypes
In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 2 columns):
A      1000 non-null object
B      1000 non-null category
dtypes: category(1), object(1)
memory usage: 9.0+ KB
```

(continues on next page)

- Index now has a `fillna` method ([GH10089](#))

- Series of type `category` now make `.str.<...>` and `.dt.<...>` accessor methods / properties available, if the categories are of that type. (GH10661)

(continues on next page)

(continued from previous page)

```

0      1
1      2
2      3
3      4
4      5
dtype: int64

```

- `pivot_table` now has a `margins_name` argument so you can use something other than the default of 'All' ([GH3335](#))
- Implement export of `datetime64[ns, tz]` dtypes with a fixed HDF5 store ([GH11411](#))
- Pretty printing sets (e.g. in `DataFrame` cells) now uses set literal syntax (`{x, y}`) instead of Legacy Python syntax (`set([x, y])`) ([GH11215](#))
- Improve the error message in `pandas.io.gbq.to_gbq()` when a streaming insert fails ([GH11285](#)) and when the `DataFrame` does not match the schema of the destination table ([GH11359](#))

### 1.15.3 API changes

- raise `NotImplementedError` in `Index.shift` for non-supported index types ([GH8038](#))
- `min` and `max` reductions on `datetime64` and `timedelta64` dtyped series now result in `NaT` and not `nan` ([GH11245](#)).
- Indexing with a null key will raise a `TypeError`, instead of a `ValueError` ([GH11356](#))
- `Series.ptp` will now ignore missing values by default ([GH11163](#))

#### 1.15.3.1 Deprecations

- The `pandas.io.ga` module which implements google-analytics support is deprecated and will be removed in a future version ([GH11308](#))
- Deprecate the `engine` keyword in `.to_csv()`, which will be removed in a future version ([GH11274](#))

### 1.15.4 Performance Improvements

- Checking monotonic-ness before sorting on an index ([GH11080](#))
- `Series.dropna` performance improvement when its dtype can't contain `NaN` ([GH11159](#))
- Release the GIL on most datetime field operations (e.g. `DatetimeIndex.year`, `Series.dt.year`), normalization, and conversion to and from `Period`, `DatetimeIndex.to_period` and `PeriodIndex.to_timestamp` ([GH11263](#))
- Release the GIL on some rolling algos: `rolling_median`, `rolling_mean`, `rolling_max`, `rolling_min`, `rolling_var`, `rolling_kurt`, `rolling_skew` ([GH11450](#))
- Release the GIL when reading and parsing text files in `read_csv`, `read_table` ([GH11272](#))
- Improved performance of `rolling_median` ([GH11450](#))
- Improved performance of `to_excel` ([GH11352](#))
- Performance bug in repr of Categorical categories, which was rendering the strings before chopping them for display ([GH11305](#))

- Performance improvement in `Categorical.remove_unused_categories`, ([GH11643](#)).
- Improved performance of `Series` constructor with no data and `DatetimeIndex` ([GH11433](#))
- Improved performance of `shift`, `cumprod`, and `cumsum` with `groupby` ([GH4095](#))

### 1.15.5 Bug Fixes

- `SparseArray.__iter__()` now does not cause `PendingDeprecationWarning` in Python 3.5 ([GH11622](#))
- Regression from 0.16.2 for output formatting of long floats/nan, restored in ([GH11302](#))
- `Series.sort_index()` now correctly handles the `inplace` option ([GH11402](#))
- Incorrectly distributed `.c` file in the build on PyPi when reading a csv of floats and passing `na_values=<a scalar>` would show an exception ([GH11374](#))
- Bug in `.to_latex()` output broken when the index has a name ([GH10660](#))
- Bug in `HDFStore.append` with strings whose encoded length exceeded the max unencoded length ([GH11234](#))
- Bug in merging `datetime64[ns, tz]` dtypes ([GH11405](#))
- Bug in `HDFStore.select` when comparing with a numpy scalar in a `where` clause ([GH11283](#))
- Bug in using `DataFrame.ix` with a multi-index indexer ([GH11372](#))
- Bug in `date_range` with ambiguous endpoints ([GH11626](#))
- Prevent adding new attributes to the accessors `.str`, `.dt` and `.cat`. Retrieving such a value was not possible, so error out on setting it. ([GH10673](#))
- Bug in tz-conversions with an ambiguous time and `.dt` accessors ([GH11295](#))
- Bug in output formatting when using an index of ambiguous times ([GH11619](#))
- Bug in comparisons of `Series` vs list-likes ([GH11339](#))
- Bug in `DataFrame.replace` with a `datetime64[ns, tz]` and a non-compatible `to_replace` ([GH11326](#), [GH11153](#))
- Bug in `isnull` where `numpy.datetime64('NaT')` in a `numpy.array` was not determined to be null ([GH11206](#))
- Bug in list-like indexing with a mixed-integer `Index` ([GH11320](#))
- Bug in `pivot_table` with `margins=True` when indexes are of `Categorical` dtype ([GH10993](#))
- Bug in `DataFrame.plot` cannot use hex strings colors ([GH10299](#))
- Regression in `DataFrame.drop_duplicates` from 0.16.2, causing incorrect results on integer values ([GH11376](#))
- Bug in `pd.eval` where unary ops in a list error ([GH11235](#))
- Bug in `squeeze()` with zero length arrays ([GH11230](#), [GH8999](#))
- Bug in `describe()` dropping column names for hierarchical indexes ([GH11517](#))
- Bug in `DataFrame.pct_change()` not propagating `axis` keyword on `.fillna` method ([GH11150](#))
- Bug in `.to_csv()` when a mix of integer and string column names are passed as the `columns` parameter ([GH11637](#))
- Bug in indexing with a `range`, ([GH11652](#))

- Bug in inference of numpy scalars and preserving dtype when setting columns (GH11638)
- Bug in `to_sql` using unicode column names giving `UnicodeEncodeError` with (GH11431).
- Fix regression in setting of `xticks` in `plot` (GH11529).
- Bug in `holiday.dates` where observance rules could not be applied to holiday and doc enhancement (GH11477, GH11533)
- Fix plotting issues when having plain `Axes` instances instead of `SubplotAxes` (GH11520, GH11556).
- Bug in `DataFrame.to_latex()` produces an extra rule when `header=False` (GH7124)
- Bug in `df.groupby(...).apply(func)` when a func returns a `Series` containing a new datetimelike column (GH11324)
- Bug in `pandas.json` when file to load is big (GH11344)
- Bugs in `to_excel` with duplicate columns (GH11007, GH10982, GH10970)
- Fixed a bug that prevented the construction of an empty series of dtype `datetime64[ns, tz]` (GH11245).
- Bug in `read_excel` with multi-index containing integers (GH11317)
- Bug in `to_excel` with `openpyxl` 2.2+ and merging (GH11408)
- Bug in `DataFrame.to_dict()` produces a `np.datetime64` object instead of `Timestamp` when only datetime is present in data (GH11327)
- Bug in `DataFrame.corr()` raises exception when computes Kendall correlation for `DataFrames` with boolean and not boolean columns (GH11560)
- Bug in the link-time error caused by `C inline` functions on FreeBSD 10+ (with `clang`) (GH10510)
- Bug in `DataFrame.to_csv` in passing through arguments for formatting `MultiIndexes`, including `date_format` (GH7791)
- Bug in `DataFrame.join()` with `how='right'` producing a `TypeError` (GH11519)
- Bug in `Series.quantile` with empty list results has `Index` with `object` dtype (GH11588)
- Bug in `pd.merge` results in empty `Int64Index` rather than `Index(dtype=object)` when the merge result is empty (GH11588)
- Bug in `Categorical.remove_unused_categories` when having `NaN` values (GH11599)
- Bug in `DataFrame.to_sparse()` loses column names for `MultiIndexes` (GH11600)
- Bug in `DataFrame.round()` with non-unique column index producing a Fatal Python error (GH11611)
- Bug in `DataFrame.round()` with `decimals` being a non-unique indexed `Series` producing extra columns (GH11618)

## 1.16 v0.17.0 (October 9, 2015)

This is a major release from 0.16.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

**Warning:** pandas >= 0.17.0 will no longer support compatibility with Python version 3.2 (GH9118)

**Warning:** The `pandas.io.data` package is deprecated and will be replaced by the `pandas-datareader` package. This will allow the data modules to be independently updated to your pandas installation. The API for `pandas-datareader v0.1.1` is exactly the same as in `pandas v0.17.0` (GH8961, GH10861).

After installing `pandas-datareader`, you can easily change your imports:

```
from pandas.io import data, wb
```

becomes

```
from pandas_datareader import data, wb
```

Highlights include:

- Release the Global Interpreter Lock (GIL) on some cython operations, see [here](#)
- Plotting methods are now available as attributes of the `.plot` accessor, see [here](#)
- The sorting API has been revamped to remove some long-time inconsistencies, see [here](#)
- Support for a `datetime64[ns]` with timezones as a first-class dtype, see [here](#)
- The default for `to_datetime` will now be to `raise` when presented with unparseable formats, previously this would return the original input. Also, date parse functions now return consistent results. See [here](#)
- The default for `dropna` in `HDFStore` has changed to `False`, to store by default all rows even if they are all `NaN`, see [here](#)
- Datetime accessor (`dt`) now supports `Series.dt.strftime` to generate formatted strings for datetime-likes, and `Series.dt.total_seconds` to generate each duration of the `timedelta` in seconds. See [here](#)
- `Period` and `PeriodIndex` can handle multiplied freq like `3D`, which corresponding to 3 days span. See [here](#)
- Development installed versions of pandas will now have PEP440 compliant version strings (GH9518)
- Development support for benchmarking with the [Air Speed Velocity](#) library (GH8361)
- Support for reading SAS `xport` files, see [here](#)
- Documentation comparing SAS to *pandas*, see [here](#)
- Removal of the automatic `TimeSeries` broadcasting, deprecated since 0.8.0, see [here](#)
- Display format with plain text can optionally align with Unicode East Asian Width, see [here](#)
- Compatibility with Python 3.5 (GH11097)
- Compatibility with matplotlib 1.5.0 (GH11111)

Check the [API Changes](#) and [deprecations](#) before updating.

### What's new in v0.17.0

- *New features*
  - *Datetime with TZ*
  - *Releasing the GIL*
  - *Plot submethods*
  - *Additional methods for `dt` accessor*
    - \* *`strftime`*

- \* *total\_seconds*
  - *Period Frequency Enhancement*
  - *Support for SAS XPORT files*
  - *Support for Math Functions in .eval()*
  - *Changes to Excel with MultiIndex*
  - *Google BigQuery Enhancements*
  - *Display Alignment with Unicode East Asian Width*
  - *Other enhancements*
- *Backwards incompatible API changes*
  - *Changes to sorting API*
  - *Changes to to\_datetime and to\_timedelta*
    - \* *Error handling*
    - \* *Consistent Parsing*
  - *Changes to Index Comparisons*
  - *Changes to Boolean Comparisons vs. None*
  - *HDFStore dropna behavior*
  - *Changes to display.precision option*
  - *Changes to Categorical.unique*
  - *Changes to bool passed as header in Parsers*
  - *Other API Changes*
  - *Deprecations*
  - *Removal of prior version deprecations/changes*
- *Performance Improvements*
- *Bug Fixes*

## 1.16.1 New features

### 1.16.1.1 Datetime with TZ

We are adding an implementation that natively supports datetime with timezones. A Series or a DataFrame column previously *could* be assigned a datetime with timezones, and would work as an object dtype. This had performance issues with a large number rows. See the [docs](#) for more details. ([GH8260](#), [GH10763](#), [GH11034](#)).

The new implementation allows for having a single-timezone across all rows, with operations in a performant manner.

```
In [1]: df = DataFrame({'A' : date_range('20130101', periods=3),
...:                  'B' : date_range('20130101', periods=3, tz='US/Eastern'),
...:                  'C' : date_range('20130101', periods=3, tz='CET')})
...:

In [2]: df
```

(continues on next page)

(continued from previous page)

```
Out [2]:
```

	A	B	C
0	2013-01-01 2013-01-01 00:00:00-05:00	2013-01-01 00:00:00+01:00	
1	2013-01-02 2013-01-02 00:00:00-05:00	2013-01-02 00:00:00+01:00	
2	2013-01-03 2013-01-03 00:00:00-05:00	2013-01-03 00:00:00+01:00	

```
In [3]: df.dtypes
```

```

////////////////////////////////////
↪
A          datetime64[ns]
B    datetime64[ns, US/Eastern]
C          datetime64[ns, CET]
dtype: object
```

```
In [4]: df.B
```

```
Out [4]:
```

0	2013-01-01 00:00:00-05:00
1	2013-01-02 00:00:00-05:00
2	2013-01-03 00:00:00-05:00

Name: B, dtype: datetime64[ns, US/Eastern]

```
In [5]: df.B.dt.tz_localize(None)
```

```

////////////////////////////////////
↪
0    2013-01-01
1    2013-01-02
2    2013-01-03
Name: B, dtype: datetime64[ns]
```

This uses a new-dtype representation as well, that is very similar in look-and-feel to its numpy cousin `datetime64[ns]`

```
In [6]: df['B'].dtype
```

```
Out [6]: datetime64[ns, US/Eastern]
```

```
In [7]: type(df['B'].dtype)
```

```
Out [7]: pandas.core.dtypes.dtypes.DatetimeTZDtype
```

**Note:** There is a slightly different string repr for the underlying `DatetimeIndex` as a result of the dtype changes, but functionally these are the same.

Previous Behavior:

```
In [1]: pd.date_range('20130101', periods=3, tz='US/Eastern')
Out [1]: DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
                        '2013-01-03 00:00:00-05:00'],
                        dtype='datetime64[ns]', freq='D', tz='US/Eastern')
```

```
In [2]: pd.date_range('20130101', periods=3, tz='US/Eastern').dtype
```

```
Out [2]: dtype('<M8[ns]')
```

New Behavior:

```
In [8]: pd.date_range('20130101', periods=3, tz='US/Eastern')
```

```
Out [8]:
```

(continues on next page)



(continued from previous page)

```
DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
              '2013-01-03 00:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

```
In [9]: pd.date_range('20130101', periods=3, tz='US/Eastern').dtype
```

```
dtype='datetime64[ns, US/Eastern]'
```

### 1.16.1.2 Releasing the GIL

We are releasing the global-interpreter-lock (GIL) on some cython operations. This will allow other threads to run simultaneously during computation, potentially allowing performance improvements from multi-threading. Notably `groupby`, `nsmallest`, `value_counts` and some indexing operations benefit from this. (GH8882)

For example the `groupby` expression in the following code will have the GIL released during the factorization step, e.g. `df.groupby('key')` as well as the `.sum()` operation.

```
N = 1000000
ngroups = 10
df = DataFrame({'key' : np.random.randint(0, ngroups, size=N),
               'data' : np.random.randn(N) })
df.groupby('key')['data'].sum()
```

Releasing of the GIL could benefit an application that uses threads for user interactions (e.g. [QT](#)), or performing multi-threaded computations. A nice example of a library that can handle these types of computation-in-parallel is the [dask](#) library.

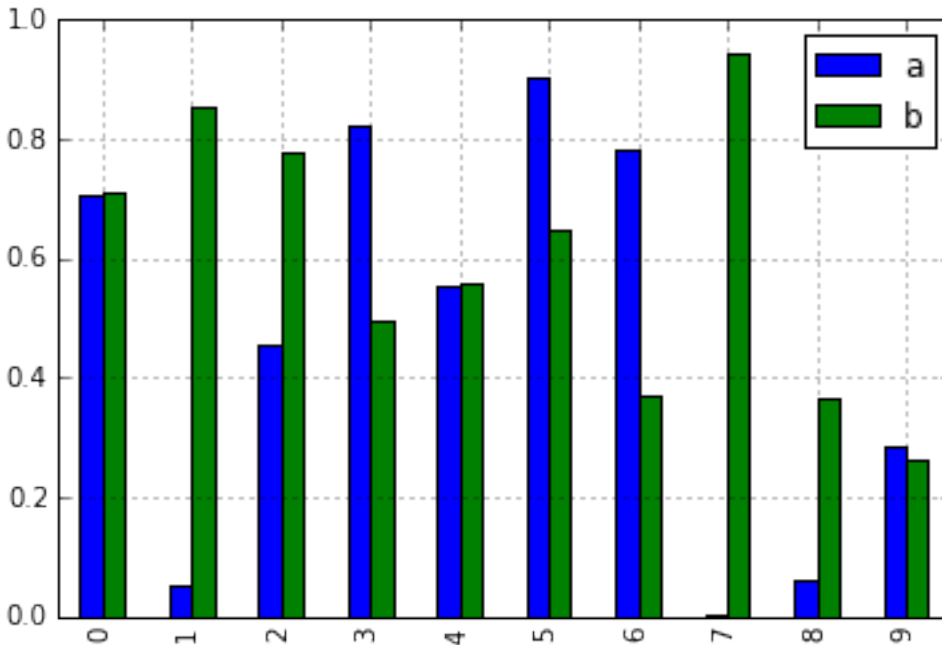
### 1.16.1.3 Plot submethods

The `Series` and `DataFrame` `.plot()` method allows for customizing *plot types* by supplying the `kind` keyword arguments. Unfortunately, many of these kinds of plots use different required and optional keyword arguments, which makes it difficult to discover what any given plot kind uses out of the dozens of possible arguments.

To alleviate this issue, we have added a new, optional plotting interface, which exposes each kind of plot as a method of the `.plot` attribute. Instead of writing `series.plot(kind=<kind>, ...)`, you can now also use `series.plot.<kind>(...)`:

```
In [10]: df = pd.DataFrame(np.random.rand(10, 2), columns=['a', 'b'])
```

```
In [11]: df.plot.bar()
```



As a result of this change, these methods are now all discoverable via tab-completion:

```
In [12]: df.plot.<TAB>
df.plot.area      df.plot.barh      df.plot.density  df.plot.hist      df.plot.line
df.plot.scatter
df.plot.bar       df.plot.box       df.plot.hexbin   df.plot.kde       df.plot.pie
```

Each method signature only includes relevant arguments. Currently, these are limited to required arguments, but in the future these will include optional arguments, as well. For an overview, see the new [Plotting API](#) documentation.

#### 1.16.1.4 Additional methods for dt accessor

##### strftime

We are now supporting a `Series.dt.strftime` method for datetime-likes to generate a formatted string (GH10110). Examples:

```
# DatetimeIndex
In [13]: s = pd.Series(pd.date_range('20130101', periods=4))

In [14]: s
Out[14]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: datetime64[ns]

In [15]: s.dt.strftime('%Y/%m/%d')
Out[15]:
0    2013/01/01
```

(continues on next page)

(continued from previous page)

```

1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object

```

```

# PeriodIndex
In [16]: s = pd.Series(pd.period_range('20130101', periods=4))

In [17]: s
Out[17]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: object

In [18]: s.dt.strftime('%Y/%m/%d')
Out[18]:
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object

```

The string format is as the python standard library and details can be found [here](#)

### total\_seconds

`pd.Series` of type `timedelta64` has new method `.dt.total_seconds()` returning the duration of the `timedelta` in seconds ([GH10817](#))

```

# TimedeltaIndex
In [19]: s = pd.Series(pd.timedelta_range('1 minutes', periods=4))

In [20]: s
Out[20]:
0    0 days 00:01:00
1    1 days 00:01:00
2    2 days 00:01:00
3    3 days 00:01:00
dtype: timedelta64[ns]

In [21]: s.dt.total_seconds()
Out[21]:
0         60.0
1    86460.0
2   172860.0
3   259260.0
dtype: float64

```

### 1.16.1.5 Period Frequency Enhancement

`Period`, `PeriodIndex` and `period_range` can now accept multiplied freq. Also, `Period.freq` and `PeriodIndex.freq` are now stored as a `DateOffset` instance like `DatetimeIndex`, and not as `str` (GH7811)

A multiplied freq represents a span of corresponding length. The example below creates a period of 3 days. Addition and subtraction will shift the period by its span.

```
In [22]: p = pd.Period('2015-08-01', freq='3D')

In [23]: p
Out[23]: Period('2015-08-01', '3D')

In [24]: p + 1
Out[24]: Period('2015-08-04', '3D')

In [25]: p - 2
Out[25]: Period('2015-07-26', '3D')

In [26]: p.to_timestamp()
Out[26]: Timestamp('2015-08-01 00:00:00')

In [27]: p.to_timestamp(how='E')
Out[27]: Timestamp('2015-08-03 00:00:00')
```

You can use the multiplied freq in `PeriodIndex` and `period_range`.

```
In [28]: idx = pd.period_range('2015-08-01', periods=4, freq='2D')

In [29]: idx
Out[29]: PeriodIndex(['2015-08-01', '2015-08-03', '2015-08-05', '2015-08-07'], dtype=
    <period[2D]', freq='2D')

In [30]: idx + 1
Out[30]: PeriodIndex(['2015-08-03', '2015-08-05', '2015-08-07', '2015-08-09'], dtype=
    <period[2D]', freq='2D')
```

### 1.16.1.6 Support for SAS XPORT files

`read_sas()` provides support for reading *SAS XPORT* format files. (GH4052).

```
df = pd.read_sas('sas_xport.xpt')
```

It is also possible to obtain an iterator and read an XPORT file incrementally.

```
for df in pd.read_sas('sas_xport.xpt', chunksize=10000):
    do_something(df)
```

See the *docs* for more details.

### 1.16.1.7 Support for Math Functions in `.eval()`

`eval()` now supports calling math functions ([GH4893](#))

```
df = pd.DataFrame({'a': np.random.randn(10)})
df.eval("b = sin(a)")
```

The support math functions are *sin*, *cos*, *exp*, *log*, *expm1*, *log1p*, *sqrt*, *sinh*, *cosh*, *tanh*, *arcsin*, *arccos*, *arctan*, *arccosh*, *arcsinh*, *arctanh*, *abs* and *arctan2*.

These functions map to the intrinsics for the NumExpr engine. For the Python engine, they are mapped to NumPy calls.

### 1.16.1.8 Changes to Excel with `MultiIndex`

In version 0.16.2 a `DataFrame` with `MultiIndex` columns could not be written to Excel via `to_excel`. That functionality has been added ([GH10564](#)), along with updating `read_excel` so that the data can be read back with, no loss of information, by specifying which columns/rows make up the `MultiIndex` in the `header` and `index_col` parameters ([GH4679](#))

See the [documentation](#) for more details.

```
In [31]: df = pd.DataFrame([[1,2,3,4], [5,6,7,8]],
.....:                    columns = pd.MultiIndex.from_product(['foo', 'bar'], ['a', 'b
↳ ']),
.....:                    names = ['col1', 'col2
↳ ']),
.....:                    index = pd.MultiIndex.from_product(['j'], ['l', 'k']),
.....:                    names = ['i1', 'i2']))

In [32]: df
Out[32]:
col1 foo    bar
col2  a  b   a  b
i1 i2
j  l    1  2   3  4
   k    5  6   7  8

In [33]: df.to_excel('test.xlsx')

In [34]: df = pd.read_excel('test.xlsx', header=[0,1], index_col=[0,1])

In [35]: df
Out[35]:
col1 foo    bar
col2  a  b   a  b
i1 i2
j  l    1  2   3  4
   k    5  6   7  8
```

Previously, it was necessary to specify the `has_index_names` argument in `read_excel`, if the serialized data had index names. For version 0.17.0 the output format of `to_excel` has been changed to make this keyword unnecessary - the change is shown below.

**Old**

	A	B	C	D	E	F
1		A	B	C	D	
2	idx_name					
3	2000-01-07 00:00:00	0.968129	0.906529	0.05343	0.02619	
4	2000-01-10 00:00:00	-0.16632	1.981993	1.833093	0.803685	
5	2000-01-11 00:00:00	0.121057	0.36946	-0.02888	1.683975	
6	2000-01-12 00:00:00	-1.70456	-0.73098	-0.38088	0.020946	
7	2000-01-13 00:00:00	-1.20024	1.907733	0.629318	1.507033	
8	2000-01-14 00:00:00	-0.66344	0.073188	1.583482	0.735205	
9	2000-01-17 00:00:00	0.716635	-2.07952	1.760536	0.970309	

New

	A	B	C	D	E	
1	idx_name	A	B	C	D	
2	2000-01-07 00:00:00	0.968129	0.906529	0.05343	0.02619	
3	2000-01-10 00:00:00	-0.16632	1.981993	1.833093	0.803685	
4	2000-01-11 00:00:00	0.121057	0.36946	-0.02888	1.683975	
5	2000-01-12 00:00:00	-1.70456	-0.73098	-0.38088	0.020946	
6	2000-01-13 00:00:00	-1.20024	1.907733	0.629318	1.507033	
7	2000-01-14 00:00:00	-0.66344	0.073188	1.583482	0.735205	
8	2000-01-17 00:00:00	0.716635	-2.07952	1.760536	0.970309	
9	2000-01-18 00:00:00	0.727628	2.22267	2.706276	0.681842	

**Warning:** Excel files saved in version 0.16.2 or prior that had index names will still be able to be read in, but the `has_index_names` argument must be specified to `True`.

### 1.16.1.9 Google BigQuery Enhancements

- Added ability to automatically create a table/dataset using the `pandas.io.gbq.to_gbq()` function if the destination table/dataset does not exist. (GH8325, GH11121).
- Added ability to replace an existing table and schema when calling the `pandas.io.gbq.to_gbq()` function via the `if_exists` argument. See the [docs](#) for more details (GH8325).
- `InvalidColumnOrder` and `InvalidPageToken` in the `gbq` module will raise `ValueError` instead of `IOError`.
- The `generate_bq_schema()` function is now deprecated and will be removed in a future version (GH11121).
- The `gbq` module will now support Python 3 (GH11094).

### 1.16.1.10 Display Alignment with Unicode East Asian Width

**Warning:** Enabling this option will affect the performance for printing of `DataFrame` and `Series` (about 2 times slower). Use only when it is actually required.

Some East Asian countries use Unicode characters its width is corresponding to 2 alphabets. If a `DataFrame` or `Series` contains these characters, the default output cannot be aligned properly. The following options are added to enable precise handling for these characters.

- `display.unicode.east_asian_width`: Whether to use the Unicode East Asian Width to calculate the display text width. (GH2612)
- `display.unicode.ambiguous_as_wide`: Whether to handle Unicode characters belong to Ambiguous as Wide. (GH11102)

```
In [36]: df = pd.DataFrame({'u': ['UK', u''], u': ['Alice', u'']})
```

```
In [37]: df;
```

```
>>> df = pd.DataFrame({'u'国籍': ['UK', u'日本'], u'名前': ['Alice', u'しのぶ']})
>>> df
      名前  国籍
0  Alice  UK
1  のぶ  日本
```

```
In [38]: pd.set_option('display.unicode.east_asian_width', True)
```

```
In [39]: df;
```

```
>>> pd.set_option('display.unicode.east_asian_width', True)
>>> df
      名前  国籍
0  Alice  UK
1  のぶ  日本
```

For further details, see [here](#)

### 1.16.1.11 Other enhancements

- Support for `openpyxl`  $\geq 2.2$ . The API for style support is now stable (GH10125)
- `merge` now accepts the argument `indicator` which adds a Categorical-type column (by default called `_merge`) to the output object that takes on the values (GH8790)

Observation Origin	<code>_merge</code> value
Merge key only in 'left' frame	<code>left_only</code>
Merge key only in 'right' frame	<code>right_only</code>
Merge key in both frames	<code>both</code>

```
In [40]: df1 = pd.DataFrame({'coll': [0, 1], 'col_left': ['a', 'b']})
```

(continues on next page)

(continued from previous page)

```
In [41]: df2 = pd.DataFrame({'coll': [1, 2, 2], 'col_right': [2, 2, 2]})

In [42]: pd.merge(df1, df2, on='coll', how='outer', indicator=True)
Out[42]:
```

	coll	col_left	col_right	_merge
0	0	a	NaN	left_only
1	1	b	2.0	both
2	2	NaN	2.0	right_only
3	2	NaN	2.0	right_only

For more, see the [updated docs](#)

- `pd.to_numeric` is a new function to coerce strings to numbers (possibly with coercion) ([GH11133](#))
- `pd.merge` will now allow duplicate column names if they are not merged upon ([GH10639](#)).
- `pd.pivot` will now allow passing index as `None` ([GH3962](#)).
- `pd.concat` will now use existing Series names if provided ([GH10698](#)).

```
In [43]: foo = pd.Series([1, 2], name='foo')

In [44]: bar = pd.Series([1, 2])

In [45]: baz = pd.Series([4, 5])
```

Previous Behavior:

```
In [1] pd.concat([foo, bar, baz], 1)
Out[1]:
```

	0	1	2
0	1	1	4
1	2	2	5

New Behavior:

```
In [46]: pd.concat([foo, bar, baz], 1)
Out[46]:
```

	foo	0	1
0	1	1	4
1	2	2	5

- `DataFrame` has gained the `nlargest` and `nsmallest` methods ([GH10393](#))
- Add a `limit_direction` keyword argument that works with `limit` to enable interpolate to fill NaN values forward, backward, or both ([GH9218](#), [GH10420](#), [GH11115](#))

```
In [47]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13])

In [48]: ser.interpolate(limit=1, limit_direction='both')
Out[48]:
```

0	NaN
1	5.0
2	5.0
3	7.0
4	NaN
5	11.0
6	13.0

dtype: float64



235

```

In [57]: df = pd.DataFrame({'x': range(5),
.....:                     't': pd.date_range('2000-01-01', periods=5)})
.....:

In [58]: df.reindex([0.1, 1.9, 3.5],
.....:               method='nearest',
.....:               tolerance=0.2)
.....:
Out[58]:
      x      t
0.1  0.0 2000-01-01
1.9  2.0 2000-01-03
3.5  NaN      NaT

```

When used on a `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`, `tolerance` will be coerced into a `Timedelta` if possible. This allows you to specify tolerance with a string:

```

In [59]: df = df.set_index('t')

In [60]: df.reindex(pd.to_datetime(['1999-12-31']),
.....:               method='nearest',
.....:               tolerance='1 day')
.....:
Out[60]:
      x
1999-12-31  0

```

`tolerance` is also exposed by the lower level `Index.get_indexer` and `Index.get_loc` methods.

- Added functionality to use the `base` argument when resampling a `TimeDeltaIndex` (GH10530)
- `DatetimeIndex` can be instantiated using strings that contain `NaT` (GH7599)
- `to_datetime` can now accept the `yearfirst` keyword (GH7599)
- `pandas.tseries.offsets` larger than the `Day` offset can now be used with a `Series` for addition/subtraction (GH10699). See the *docs* for more details.
- `pd.Timedelta.total_seconds()` now returns `Timedelta` duration to ns precision (previously microsecond precision) (GH10939)
- `PeriodIndex` now supports arithmetic with `np.ndarray` (GH10638)
- Support pickling of `Period` objects (GH10439)
- `.as_blocks` will now take a `copy` optional argument to return a copy of the data, default is to copy (no change in behavior from prior versions), (GH9607)
- `regex` argument to `DataFrame.filter` now handles numeric column names instead of raising `ValueError` (GH10384).
- Enable reading gzip compressed files via URL, either by explicitly setting the `compression` parameter or by inferring from the presence of the `HTTP Content-Encoding` header in the response (GH8685)
- Enable writing Excel files in *memory* using `StringIO/BytesIO` (GH7074)
- Enable serialization of lists and dicts to strings in `ExcelWriter` (GH8188)
- SQL io functions now accept a `SQLAlchemy` connectable. (GH7877)
- `pd.read_sql` and `to_sql` can accept database URI as `con` parameter (GH10214)
- `read_sql_table` will now allow reading from views (GH10750).

- Enable writing complex values to HDFStores when using the table format ([GH10447](#))
- Enable `pd.read_hdf` to be used without specifying a key when the HDF file contains a single dataset ([GH10443](#))
- `pd.read_stata` will now read Stata 118 type files. ([GH9882](#))
- `msgpack` submodule has been updated to 0.4.6 with backward compatibility ([GH10581](#))
- `DataFrame.to_dict` now accepts `orient='index'` keyword argument ([GH10844](#)).
- `DataFrame.apply` will return a Series of dicts if the passed function returns a dict and `reduce=True` ([GH8735](#)).
- Allow passing *kwargs* to the interpolation methods ([GH10378](#)).
- Improved error message when concatenating an empty iterable of Dataframe objects ([GH9157](#))
- `pd.read_csv` can now read bz2-compressed files incrementally, and the C parser can read bz2-compressed files from AWS S3 ([GH11070](#), [GH11072](#)).
- In `pd.read_csv`, recognize `s3n://` and `s3a://` URLs as designating S3 file storage ([GH11070](#), [GH11071](#)).
- Read CSV files from AWS S3 incrementally, instead of first downloading the entire file. (Full file download still required for compressed files in Python 2.) ([GH11070](#), [GH11073](#))
- `pd.read_csv` is now able to infer compression type for files read from AWS S3 storage ([GH11070](#), [GH11074](#)).

## 1.16.2 Backwards incompatible API changes

### 1.16.2.1 Changes to sorting API

The sorting API has had some longtime inconsistencies. ([GH9816](#), [GH8239](#)).

Here is a summary of the API **PRIOR** to 0.17.0:

- `Series.sort` is **INPLACE** while `DataFrame.sort` returns a new object.
- `Series.order` returns a new object
- It was possible to use `Series/DataFrame.sort_index` to sort by **values** by passing the `by` keyword.
- `Series/DataFrame.sortlevel` worked only on a `MultiIndex` for sorting by index.

To address these issues, we have revamped the API:

- We have introduced a new method, `DataFrame.sort_values()`, which is the merger of `DataFrame.sort()`, `Series.sort()`, and `Series.order()`, to handle sorting of **values**.
- The existing methods `Series.sort()`, `Series.order()`, and `DataFrame.sort()` have been deprecated and will be removed in a future version.
- The `by` argument of `DataFrame.sort_index()` has been deprecated and will be removed in a future version.
- The existing method `.sort_index()` will gain the `level` keyword to enable level sorting.

We now have two distinct and non-overlapping methods of sorting. A \* marks items that will show a `FutureWarning`.

To sort by the **values**:

Previous	Replacement
* Series.order()	Series.sort_values()
* Series.sort()	Series.sort_values(inplace=True)
* DataFrame.sort(columns=...)	DataFrame.sort_values(by=...)

To sort by the **index**:

Previous	Replacement
Series.sort_index()	Series.sort_index()
Series.sortlevel(level=...)	Series.sort_index(level=...)
DataFrame.sort_index()	DataFrame.sort_index()
DataFrame.sortlevel(level=...)	DataFrame.sort_index(level=...)
* DataFrame.sort()	DataFrame.sort_index()

We have also deprecated and changed similar methods in two Series-like classes, Index and Categorical.

Previous	Replacement
* Index.order()	Index.sort_values()
* Categorical.order()	Categorical.sort_values()

### 1.16.2.2 Changes to to\_datetime and to\_timedelta

#### Error handling

The default for `pd.to_datetime` error handling has changed to `errors='raise'`. In prior versions it was `errors='ignore'`. Furthermore, the `coerce` argument has been deprecated in favor of `errors='coerce'`. This means that invalid parsing will raise rather than return the original input as in previous versions. ([GH10636](#))

Previous Behavior:

```
In [2]: pd.to_datetime(['2009-07-31', 'asd'])
Out[2]: array(['2009-07-31', 'asd'], dtype=object)
```

New Behavior:

```
In [3]: pd.to_datetime(['2009-07-31', 'asd'])
ValueError: Unknown string format
```

Of course you can coerce this as well.

```
In [61]: to_datetime(['2009-07-31', 'asd'], errors='coerce')
Out[61]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```

To keep the previous behavior, you can use `errors='ignore'`:

```
In [62]: to_datetime(['2009-07-31', 'asd'], errors='ignore')
Out[62]: array(['2009-07-31', 'asd'], dtype=object)
```

Furthermore, `pd.to_timedelta` has gained a similar API, of `errors='raise'|'ignore'|'coerce'`, and the `coerce` keyword has been deprecated in favor of `errors='coerce'`.

## Consistent Parsing

The string parsing of `to_datetime`, `Timestamp` and `DatetimeIndex` has been made consistent. ([GH7599](#))

Prior to v0.17.0, `Timestamp` and `to_datetime` may parse year-only datetime-string incorrectly using today's date, otherwise `DatetimeIndex` uses the beginning of the year. `Timestamp` and `to_datetime` may raise `ValueError` in some types of datetime-string which `DatetimeIndex` can parse, such as a quarterly string.

Previous Behavior:

```
In [1]: Timestamp('2012Q2')
Traceback
...
ValueError: Unable to parse 2012Q2

# Results in today's date.
In [2]: Timestamp('2014')
Out [2]: 2014-08-12 00:00:00
```

v0.17.0 can parse them as below. It works on `DatetimeIndex` also.

New Behavior:

```
In [63]: Timestamp('2012Q2')
Out [63]: Timestamp('2012-04-01 00:00:00')

In [64]: Timestamp('2014')
Out [64]: Timestamp('2014-01-01 00:00:00')

In [65]: DatetimeIndex(['2012Q2', '2014'])
Out [65]: DatetimeIndex(['2012-04-01', '2014-01-01'], dtype='datetime64[ns]', freq=None)
```

**Note:** If you want to perform calculations based on today's date, use `Timestamp.now()` and `pandas.tseries.offsets`.

```
In [66]: import pandas.tseries.offsets as offsets

In [67]: Timestamp.now()
Out [67]: Timestamp('2018-07-05 17:10:40.227744')

In [68]: Timestamp.now() + offsets.DateOffset(years=1)
Out [68]: Timestamp('2019-07-05 17:10:40.228803')
```

### 1.16.2.3 Changes to Index Comparisons

Operator equal on Index should behavior similarly to Series ([GH9947](#), [GH10637](#))

Starting in v0.17.0, comparing Index objects of different lengths will raise a `ValueError`. This is to be consistent with the behavior of Series.

Previous Behavior:

```
In [2]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[2]: array([ True, False, False], dtype=bool)

In [3]: pd.Index([1, 2, 3]) == pd.Index([2])
Out[3]: array([False,  True, False], dtype=bool)

In [4]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
Out[4]: False
```

New Behavior:

```
In [8]: pd.Index([1, 2, 3]) == pd.Index([1, 4, 5])
Out[8]: array([ True, False, False], dtype=bool)

In [9]: pd.Index([1, 2, 3]) == pd.Index([2])
ValueError: Lengths must match to compare

In [10]: pd.Index([1, 2, 3]) == pd.Index([1, 2])
ValueError: Lengths must match to compare
```

Note that this is different from the numpy behavior where a comparison can be broadcast:

```
In [69]: np.array([1, 2, 3]) == np.array([1])
Out[69]: array([ True, False, False], dtype=bool)
```

or it can return False if broadcasting can not be done:

```
In [70]: np.array([1, 2, 3]) == np.array([1, 2])
Out[70]: False
```

#### 1.16.2.4 Changes to Boolean Comparisons vs. None

Boolean comparisons of a Series vs None will now be equivalent to comparing with `np.nan`, rather than raise `TypeError`. (GH1079).

```
In [71]: s = Series(range(3))

In [72]: s.iloc[1] = None

In [73]: s
Out[73]:
0    0.0
1    NaN
2    2.0
dtype: float64
```

Previous Behavior:

```
In [5]: s==None
TypeError: Could not compare <type 'NoneType'> type with Series
```

New Behavior:

```
In [74]: s==None
Out[74]:
```

(continues on next page)

(continued from previous page)

```
0    False
1    False
2    False
dtype: bool
```

Usually you simply want to know which values are null.

```
In [75]: s.isnull()
Out[75]:
0    False
1     True
2    False
dtype: bool
```

**Warning:** You generally will want to use `isnull/notnull` for these types of comparisons, as `isnull/notnull` tells you which elements are null. One has to be mindful that `nan`'s don't compare equal, but `None`'s do. Note that Pandas/numpy uses the fact that `np.nan != np.nan`, and treats `None` like `np.nan`.

```
In [76]: None == None
Out[76]: True

In [77]: np.nan == np.nan
Out[77]: False
```

### 1.16.2.5 HDFStore dropna behavior

The default behavior for `HDFStore` write functions with `format='table'` is now to keep rows that are all missing. Previously, the behavior was to drop rows that were all missing save the index. The previous behavior can be replicated using the `dropna=True` option. (GH9382)

Previous Behavior:

```
In [78]: df_with_missing = pd.DataFrame({'col1':[0, np.nan, 2],
....:                                  'col2':[1, np.nan, np.nan]})
....:

In [79]: df_with_missing
Out[79]:
   col1  col2
0    0.0    1.0
1    NaN    NaN
2    2.0    NaN
```

```
In [27]: df_with_missing.to_hdf('file.h5',
...                             'df_with_missing',
...                             format='table',
...                             mode='w')

In [28]: pd.read_hdf('file.h5', 'df_with_missing')

Out [28]:
   col1  col2
```

(continues on next page)

(continued from previous page)

0	0	1
2	2	NaN

New Behavior:

```
In [80]: df_with_missing.to_hdf('file.h5',
.....:                        'df_with_missing',
.....:                        format='table',
.....:                        mode='w')
.....:

In [81]: pd.read_hdf('file.h5', 'df_with_missing')
Out[81]:
   col1  col2
0    0.0    1.0
1   NaN   NaN
2    2.0   NaN
```

See the [docs](#) for more details.

### 1.16.2.6 Changes to `display.precision` option

The `display.precision` option has been clarified to refer to decimal places ([GH10451](#)).

Earlier versions of pandas would format floating point numbers to have one less decimal place than the value in `display.precision`.

```
In [1]: pd.set_option('display.precision', 2)

In [2]: pd.DataFrame({'x': [123.456789]})
Out[2]:
      x
0  123.5
```

If interpreting precision as “significant figures” this did work for scientific notation but that same interpretation did not work for values with standard formatting. It was also out of step with how numpy handles formatting.

Going forward the value of `display.precision` will directly control the number of places after the decimal, for regular formatting as well as scientific notation, similar to how numpy’s `precision` print option works.

```
In [82]: pd.set_option('display.precision', 2)

In [83]: pd.DataFrame({'x': [123.456789]})
Out[83]:
      x
0  123.46
```

To preserve output behavior with prior versions the default value of `display.precision` has been reduced to 6 from 7.

### 1.16.2.7 Changes to `Categorical.unique`

`Categorical.unique` now returns new `Categoricals` with `categories` and `codes` that are unique, rather than returning `np.array` ([GH10508](#))

- `unordered` category: values and categories are sorted by appearance order.





- When constructing `DataFrame` with an array of `complex64` dtype previously meant the corresponding column was automatically promoted to the `complex128` dtype. Pandas will now preserve the itemsize of the input for complex data ([GH10952](#))
- some numeric reduction operators would return `ValueError`, rather than `TypeError` on object types that includes strings and numbers ([GH11131](#))
- Passing currently unsupported `chunksize` argument to `read_excel` or `ExcelFile.parse` will now raise `NotImplementedError` ([GH8011](#))
- Allow an `ExcelFile` object to be passed into `read_excel` ([GH11198](#))
- `DatetimeIndex.union` does not infer `freq` if self and the input have `None` as `freq` ([GH11086](#))
- `NaT`'s methods now either raise `ValueError`, or return `np.nan` or `NaT` ([GH9513](#))

Behavior	Methods
return <code>np.nan</code>	<code>weekday</code> , <code>isoweekday</code>
return <code>NaT</code>	<code>date</code> , <code>now</code> , <code>replace</code> , <code>to_datetime</code> , <code>today</code>
return <code>np.datetime64('NaT')</code>	<code>to_datetime64</code> (unchanged)
raise <code>ValueError</code>	All other public methods (names not beginning with underscores)

#### 1.16.2.10 Deprecations

- For `Series` the following indexing functions are deprecated ([GH10177](#)).

Deprecated Function	Replacement
<code>.irow(i)</code>	<code>.iloc[i]</code> or <code>.iat[i]</code>
<code>.iget(i)</code>	<code>.iloc[i]</code> or <code>.iat[i]</code>
<code>.iget_value(i)</code>	<code>.iloc[i]</code> or <code>.iat[i]</code>

- For `DataFrame` the following indexing functions are deprecated ([GH10177](#)).

Deprecated Function	Replacement
<code>.irow(i)</code>	<code>.iloc[i]</code>
<code>.iget_value(i, j)</code>	<code>.iloc[i, j]</code> or <code>.iat[i, j]</code>
<code>.icol(j)</code>	<code>.iloc[:, j]</code>

---

**Note:** These indexing function have been deprecated in the documentation since 0.11.0.

---

- `Categorical.name` was deprecated to make `Categorical` more `numpy.ndarray` like. Use `Series(cat, name="whatever")` instead ([GH10482](#)).
- Setting missing values (`NaN`) in a `Categorical`'s categories will issue a warning ([GH10748](#)). You can still have missing values in the values.
- `drop_duplicates` and `uplicated`'s `take_last` keyword was deprecated in favor of `keep`. ([GH6511](#), [GH8505](#))
- `Series.nsmallest` and `nlargest`'s `take_last` keyword was deprecated in favor of `keep`. ([GH10792](#))
- `DataFrame.combineAdd` and `DataFrame.combineMult` are deprecated. They can easily be replaced by using the `add` and `mul` methods: `DataFrame.add(other, fill_value=0)` and `DataFrame.mul(other, fill_value=1.)` ([GH10735](#)).

- `TimeSeries` deprecated in favor of `Series` (note that this has been an alias since 0.13.0), ([GH10890](#))
- `SparsePanel` deprecated and will be removed in a future version ([GH11157](#)).
- `Series.is_time_series` deprecated in favor of `Series.index.is_all_dates` ([GH11135](#))
- Legacy offsets (like `'A@JAN'`) are deprecated (note that this has been alias since 0.8.0) ([GH10878](#))
- `WidePanel` deprecated in favor of `Panel`, `LongPanel` in favor of `DataFrame` (note these have been aliases since < 0.11.0), ([GH10892](#))
- `DataFrame.convert_objects` has been deprecated in favor of type-specific functions `pd.to_datetime`, `pd.to_timestamp` and `pd.to_numeric` (new in 0.17.0) ([GH11133](#)).

#### 1.16.2.11 Removal of prior version deprecations/changes

- Removal of `na_last` parameters from `Series.order()` and `Series.sort()`, in favor of `na_position`. ([GH5231](#))
- Remove of `percentile_width` from `.describe()`, in favor of percentiles. ([GH7088](#))
- Removal of `colSpace` parameter from `DataFrame.to_string()`, in favor of `col_space`, circa 0.8.0 version.
- Removal of automatic time-series broadcasting ([GH2304](#))

```
In [90]: np.random.seed(1234)

In [91]: df = DataFrame(np.random.randn(5,2), columns=list('AB'), index=date_range(
↳ '20130101', periods=5))

In [92]: df
Out[92]:
```

	A	B
2013-01-01	0.471435	-1.190976
2013-01-02	1.432707	-0.312652
2013-01-03	-0.720589	0.887163
2013-01-04	0.859588	-0.636524
2013-01-05	0.015696	-2.242685

Previously

```
In [3]: df + df.A
FutureWarning: TimeSeries broadcasting along DataFrame index by default is_
↳ deprecated.
Please use DataFrame.<op> to explicitly broadcast arithmetic operations along the_
↳ index

Out[3]:
```

	A	B
2013-01-01	0.942870	-0.719541
2013-01-02	2.865414	1.120055
2013-01-03	-1.441177	0.166574
2013-01-04	1.719177	0.223065
2013-01-05	0.031393	-2.226989

Current

```
In [93]: df.add(df.A, axis='index')
Out [93]:
```

	A	B
2013-01-01	0.942870	-0.719541
2013-01-02	2.865414	1.120055
2013-01-03	-1.441177	0.166574
2013-01-04	1.719177	0.223065
2013-01-05	0.031393	-2.226989

- Remove `table` keyword in `HDFStore.put/append`, in favor of using `format=` (GH4645)
- Remove `kind` in `read_excel/ExcelFile` as its unused (GH4712)
- Remove `infer_type` keyword from `pd.read_html` as its unused (GH4770, GH7032)
- Remove `offset` and `timeRule` keywords from `Series.tshift/shift`, in favor of `freq` (GH4853, GH4864)
- Remove `pd.load/pd.save` aliases in favor of `pd.to_pickle/pd.read_pickle` (GH3787)

### 1.16.3 Performance Improvements

- Development support for benchmarking with the `Air Speed Velocity` library (GH8361)
- Added `vbench` benchmarks for alternative `ExcelWriter` engines and reading Excel files (GH7171)
- Performance improvements in `Categorical.value_counts` (GH10804)
- Performance improvements in `SeriesGroupBy.nunique` and `SeriesGroupBy.value_counts` and `SeriesGroupBy.transform` (GH10820, GH11077)
- Performance improvements in `DataFrame.drop_duplicates` with integer dtypes (GH10917)
- Performance improvements in `DataFrame.duplicated` with wide frames. (GH10161, GH11180)
- 4x improvement in `timedelta` string parsing (GH6755, GH10426)
- 8x improvement in `timedelta64` and `datetime64` ops (GH6755)
- Significantly improved performance of indexing `MultiIndex` with slicers (GH10287)
- 8x improvement in `iloc` using list-like input (GH10791)
- Improved performance of `Series.isin` for `datetimelike/integer` Series (GH10287)
- 20x improvement in `concat` of `Categoricals` when categories are identical (GH10587)
- Improved performance of `to_datetime` when specified format string is ISO8601 (GH10178)
- 2x improvement of `Series.value_counts` for float dtype (GH10821)
- Enable `infer_datetime_format` in `to_datetime` when date components do not have 0 padding (GH11142)
- Regression from 0.16.1 in constructing `DataFrame` from nested dictionary (GH11084)
- Performance improvements in addition/subtraction operations for `DateOffset` with `Series` or `DatetimeIndex` (GH10744, GH11205)

### 1.16.4 Bug Fixes

- Bug in incorrection computation of `.mean()` on `timedelta64[ns]` because of overflow ([GH9442](#))
- Bug in `.isin` on older numpies ([GH11232](#))
- Bug in `DataFrame.to_html(index=False)` renders unnecessary name row ([GH10344](#))
- Bug in `DataFrame.to_latex()` the `column_format` argument could not be passed ([GH9402](#))
- Bug in `DatetimeIndex` when localizing with `NaT` ([GH10477](#))
- Bug in `Series.dt` ops in preserving meta-data ([GH10477](#))
- Bug in preserving `NaT` when passed in an otherwise invalid `to_datetime` construction ([GH10477](#))
- Bug in `DataFrame.apply` when function returns categorical series. ([GH9573](#))
- Bug in `to_datetime` with invalid dates and formats supplied ([GH10154](#))
- Bug in `Index.drop_duplicates` dropping name(s) ([GH10115](#))
- Bug in `Series.quantile` dropping name ([GH10881](#))
- Bug in `pd.Series` when setting a value on an empty `Series` whose index has a frequency. ([GH10193](#))
- Bug in `pd.Series.interpolate` with invalid order keyword values. ([GH10633](#))
- Bug in `DataFrame.plot` raises `ValueError` when color name is specified by multiple characters ([GH10387](#))
- Bug in `Index` construction with a mixed list of tuples ([GH10697](#))
- Bug in `DataFrame.reset_index` when index contains `NaT`. ([GH10388](#))
- Bug in `ExcelReader` when worksheet is empty ([GH6403](#))
- Bug in `BinGrouper.group_info` where returned values are not compatible with base class ([GH10914](#))
- Bug in clearing the cache on `DataFrame.pop` and a subsequent inplace op ([GH10912](#))
- Bug in indexing with a mixed-integer `Index` causing an `ImportError` ([GH10610](#))
- Bug in `Series.count` when index has nulls ([GH10946](#))
- Bug in pickling of a non-regular freq `DatetimeIndex` ([GH11002](#))
- Bug causing `DataFrame.where` to not respect the `axis` parameter when the frame has a symmetric shape. ([GH9736](#))
- Bug in `Table.select_column` where name is not preserved ([GH10392](#))
- Bug in `offsets.generate_range` where start and end have finer precision than offset ([GH9907](#))
- Bug in `pd.rolling_*` where `Series.name` would be lost in the output ([GH10565](#))
- Bug in stack when index or columns are not unique. ([GH10417](#))
- Bug in setting a `Panel` when an axis has a multi-index ([GH10360](#))
- Bug in `USFederalHolidayCalendar` where `USMemorialDay` and `USMartinLutherKingJr` were incorrect ([GH10278](#) and [GH9760](#))
- Bug in `.sample()` where returned object, if set, gives unnecessary `SettingWithCopyWarning` ([GH10738](#))
- Bug in `.sample()` where weights passed as `Series` were not aligned along axis before being treated positionally, potentially causing problems if weight indices were not aligned with sampled object. ([GH10738](#))

- Regression fixed in ([GH9311](#), [GH6620](#), [GH9345](#)), where groupby with a datetime-like converting to float with certain aggregators ([GH10979](#))
- Bug in `DataFrame.interpolate` with `axis=1` and `inplace=True` ([GH10395](#))
- Bug in `io.sql.get_schema` when specifying multiple columns as primary key ([GH10385](#)).
- Bug in `groupby(sort=False)` with datetime-like Categorical raises `ValueError` ([GH10505](#))
- Bug in `groupby(axis=1)` with `filter()` throws `IndexError` ([GH11041](#))
- Bug in `test_categorical` on big-endian builds ([GH10425](#))
- Bug in `Series.shift` and `DataFrame.shift` not supporting categorical data ([GH9416](#))
- Bug in `Series.map` using categorical Series raises `AttributeError` ([GH10324](#))
- Bug in `MultiIndex.get_level_values` including Categorical raises `AttributeError` ([GH10460](#))
- Bug in `pd.get_dummies` with `sparse=True` not returning `SparseDataFrame` ([GH10531](#))
- Bug in Index subtypes (such as `PeriodIndex`) not returning their own type for `.drop` and `.insert` methods ([GH10620](#))
- Bug in `algos.outer_join_indexer` when right array is empty ([GH10618](#))
- Bug in `filter` (regression from 0.16.0) and `transform` when grouping on multiple keys, one of which is datetime-like ([GH10114](#))
- Bug in `to_datetime` and `to_timedelta` causing Index name to be lost ([GH10875](#))
- Bug in `len(DataFrame.groupby)` causing `IndexError` when there's a column containing only NaNs ([GH11016](#))
- Bug that caused segfault when resampling an empty Series ([GH10228](#))
- Bug in `DatetimeIndex` and `PeriodIndex.value_counts` resets name from its result, but retains in result's Index. ([GH10150](#))
- Bug in `pd.eval` using `numexpr` engine coerces 1 element numpy array to scalar ([GH10546](#))
- Bug in `pd.concat` with `axis=0` when column is of dtype category ([GH10177](#))
- Bug in `read_msgpack` where input type is not always checked ([GH10369](#), [GH10630](#))
- Bug in `pd.read_csv` with kwargs `index_col=False`, `index_col=['a', 'b']` or dtype ([GH10413](#), [GH10467](#), [GH10577](#))
- Bug in `Series.from_csv` with header kwarg not setting the `Series.name` or the `Series.index.name` ([GH10483](#))
- Bug in `groupby.var` which caused variance to be inaccurate for small float values ([GH10448](#))
- Bug in `Series.plot(kind='hist')` Y Label not informative ([GH10485](#))
- Bug in `read_csv` when using a converter which generates a `uint8` type ([GH9266](#))
- Bug causes memory leak in time-series line and area plot ([GH9003](#))
- Bug when setting a Panel sliced along the major or minor axes when the right-hand side is a `DataFrame` ([GH11014](#))
- Bug that returns `None` and does not raise `NotImplementedError` when operator functions (e.g. `.add`) of Panel are not implemented ([GH7692](#))
- Bug in line and kde plot cannot accept multiple colors when `subplots=True` ([GH9894](#))

- Bug in `DataFrame.plot` raises `ValueError` when color name is specified by multiple characters ([GH10387](#))
- Bug in left and right align of `Series` with `MultiIndex` may be inverted ([GH10665](#))
- Bug in left and right join of with `MultiIndex` may be inverted ([GH10741](#))
- Bug in `read_stata` when reading a file with a different order set in columns ([GH10757](#))
- Bug in `Categorical` may not representing properly when category contains `tz` or `Period` ([GH10713](#))
- Bug in `Categorical.__iter__` may not returning correct `datetime` and `Period` ([GH10713](#))
- Bug in indexing with a `PeriodIndex` on an object with a `PeriodIndex` ([GH4125](#))
- Bug in `read_csv` with `engine='c'`: EOF preceded by a comment, blank line, etc. was not handled correctly ([GH10728](#), [GH10548](#))
- Reading “famafrench” data via `DataReader` results in HTTP 404 error because of the website url is changed ([GH10591](#)).
- Bug in `read_msgpack` where `DataFrame` to decode has duplicate column names ([GH9618](#))
- Bug in `io.common.get_filepath_or_buffer` which caused reading of valid S3 files to fail if the bucket also contained keys for which the user does not have read permission ([GH10604](#))
- Bug in vectorised setting of timestamp columns with `python datetime.date` and `numpy datetime64` ([GH10408](#), [GH10412](#))
- Bug in `Index.take` may add unnecessary `freq` attribute ([GH10791](#))
- Bug in merge with empty `DataFrame` may raise `IndexError` ([GH10824](#))
- Bug in `to_latex` where unexpected keyword argument for some documented arguments ([GH10888](#))
- Bug in indexing of large `DataFrame` where `IndexError` is uncaught ([GH10645](#) and [GH10692](#))
- Bug in `read_csv` when using the `nrows` or `chunksize` parameters if file contains only a header line ([GH9535](#))
- Bug in serialization of category types in HDF5 in presence of alternate encodings. ([GH10366](#))
- Bug in `pd.DataFrame` when constructing an empty `DataFrame` with a string dtype ([GH9428](#))
- Bug in `pd.DataFrame.diff` when `DataFrame` is not consolidated ([GH10907](#))
- Bug in `pd.unique` for arrays with the `datetime64` or `timedelta64` dtype that meant an array with object dtype was returned instead the original dtype ([GH9431](#))
- Bug in `Timedelta` raising error when slicing from 0s ([GH10583](#))
- Bug in `DatetimeIndex.take` and `TimedeltaIndex.take` may not raise `IndexError` against invalid index ([GH10295](#))
- Bug in `Series([np.nan]).astype('M8[ms]')`, which now returns `Series([pd.NaT])` ([GH10747](#))
- Bug in `PeriodIndex.order` reset `freq` ([GH10295](#))
- Bug in `date_range` when `freq` divides `end` as `nanos` ([GH10885](#))
- Bug in `iloc` allowing memory outside bounds of a `Series` to be accessed with negative integers ([GH10779](#))
- Bug in `read_msgpack` where encoding is not respected ([GH10581](#))
- Bug preventing access to the first index when using `iloc` with a list containing the appropriate negative integer ([GH10547](#), [GH10779](#))

- Bug in `TimedeltaIndex` formatter causing error while trying to save `DataFrame` with `TimedeltaIndex` using `to_csv` (GH10833)
- Bug in `DataFrame.where` when handling `Series` slicing (GH10218, GH9558)
- Bug where `pd.read_gbq` throws `ValueError` when Bigquery returns zero rows (GH10273)
- Bug in `to_json` which was causing segmentation fault when serializing 0-rank `ndarray` (GH9576)
- Bug in plotting functions may raise `IndexError` when plotted on `GridSpec` (GH10819)
- Bug in plot result may show unnecessary minor ticklabels (GH10657)
- Bug in `groupby` incorrect computation for aggregation on `DataFrame` with `NaT` (E.g `first`, `last`, `min`). (GH10590, GH11010)
- Bug when constructing `DataFrame` where passing a dictionary with only scalar values and specifying columns did not raise an error (GH10856)
- Bug in `.var()` causing roundoff errors for highly similar values (GH10242)
- Bug in `DataFrame.plot(subplots=True)` with duplicated columns outputs incorrect result (GH10962)
- Bug in `Index` arithmetic may result in incorrect class (GH10638)
- Bug in `date_range` results in empty if `freq` is negative annually, quarterly and monthly (GH11018)
- Bug in `DatetimeIndex` cannot infer negative `freq` (GH11018)
- Remove use of some deprecated numpy comparison operations, mainly in tests. (GH10569)
- Bug in `Index` dtype may not applied properly (GH11017)
- Bug in `io.gbq` when testing for minimum google api client version (GH10652)
- Bug in `DataFrame` construction from nested dict with `timedelta` keys (GH11129)
- Bug in `.fillna` against may raise `TypeError` when data contains `datetime` dtype (GH7095, GH11153)
- Bug in `.groupby` when number of keys to group by is same as length of index (GH11185)
- Bug in `convert_objects` where converted values might not be returned if all null and `coerce` (GH9589)
- Bug in `convert_objects` where `copy` keyword was not respected (GH9589)

## 1.17 v0.16.2 (June 12, 2015)

This is a minor bug-fix release from 0.16.1 and includes a a large number of bug fixes along some new features (`pipe()` method), enhancements, and performance improvements.

We recommend that all users upgrade to this version.

Highlights include:

- A new `pipe` method, see [here](#)
- Documentation on how to use `numba` with `pandas`, see [here](#)

### What's new in v0.16.2

- *New features*
  - *Pipe*



– *Other Enhancements*

- *API Changes*
- *Performance Improvements*
- *Bug Fixes*

## 1.17.1 New features

### 1.17.1.1 Pipe

We've introduced a new method `DataFrame.pipe()`. As suggested by the name, `pipe` should be used to pipe data through a chain of function calls. The goal is to avoid confusing nested function calls like

```
# df is a DataFrame
# f, g, and h are functions that take and return DataFrames
f(g(h(df), arg1=1), arg2=2, arg3=3)
```

The logic flows from inside out, and function names are separated from their keyword arguments. This can be rewritten as

```
(df.pipe(h)
    .pipe(g, arg1=1)
    .pipe(f, arg2=2, arg3=3)
)
```

Now both the code and the logic flow from top to bottom. Keyword arguments are next to their functions. Overall the code is much more readable.

In the example above, the functions `f`, `g`, and `h` each expected the `DataFrame` as the first positional argument. When the function you wish to apply takes its data anywhere other than the first argument, pass a tuple of `(function, keyword)` indicating where the `DataFrame` should flow. For example:

```
In [1]: import statsmodels.formula.api as sm

In [2]: bb = pd.read_csv('data/baseball.csv', index_col='id')

# sm.ols takes (formula, data)
In [3]: (bb.query('h > 0')
...:     .assign(ln_h = lambda df: np.log(df.h))
...:     .pipe((sm.ols, 'data'), 'hr ~ ln_h + year + g + C(lg)')
...:     .fit()
...:     .summary()
...: )
...:
Out[3]:
<class 'statsmodels.iolib.summary.Summary'>
"""
                                OLS Regression Results
=====
Dep. Variable:                  hr      R-squared:                0.685
Model:                            OLS      Adj. R-squared:           0.665
Method:                    Least Squares   F-statistic:              34.28
Date:                Thu, 05 Jul 2018      Prob (F-statistic):       3.48e-15
Time:                  17:10:40      Log-Likelihood:          -205.92
```

(continues on next page)

(continued from previous page)

```

No. Observations:      68    AIC:      421.8
Df Residuals:         63    BIC:      432.9
Df Model:              4
Covariance Type:      nonrobust
=====
              coef      std err          t      P>|t|      [0.025      0.975]
-----
Intercept    -8484.7720    4664.146     -1.819     0.074    -1.78e+04     835.780
C(lg) [T.NL]   -2.2736     1.325     -1.716     0.091     -4.922      0.375
ln_h          -1.3542     0.875     -1.547     0.127     -3.103      0.395
year           4.2277     2.324      1.819     0.074     -0.417      8.872
g              0.1841     0.029      6.258     0.000      0.125      0.243
=====
Omnibus:          10.875    Durbin-Watson:          1.999
Prob(Omnibus) :      0.004    Jarque-Bera (JB) :      17.298
Skew:             0.537    Prob(JB) :          0.000175
Kurtosis:         5.225    Cond. No.          1.49e+07
=====

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
    specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
    strong multicollinearity or other numerical problems.
"""

```

The pipe method is inspired by unix pipes, which stream text through processes. More recently `dplyr` and `magrittr` have introduced the popular `(%>%)` pipe operator for R.

See the [documentation](#) for more. (GH10129)

### 1.17.1.2 Other Enhancements

- Added `rsplit` to Index/Series StringMethods (GH10303)
- Removed the hard-coded size limits on the DataFrame HTML representation in the IPython notebook, and leave this to IPython itself (only for IPython v3.0 or greater). This eliminates the duplicate scroll bars that appeared in the notebook with large frames (GH10231).

Note that the notebook has a `toggle output scrolling` feature to limit the display of very large frames (by clicking left of the output). You can also configure the way DataFrames are displayed using the pandas options, see here [here](#).

- `axis` parameter of `DataFrame.quantile` now accepts also `index` and `column`. (GH9543)

## 1.17.2 API Changes

- `Holiday` now raises `NotImplementedError` if both `offset` and `observance` are used in the constructor instead of returning an incorrect result (GH10217).

## 1.17.3 Performance Improvements

- Improved `Series.resample` performance with `dtype=datetime64[ns]` (GH7754)
- Increase performance of `str.split` when `expand=True` (GH10081)

### 1.17.4 Bug Fixes

- Bug in `Series.hist` raises an error when a one row `Series` was given ([GH10214](#))
- Bug where `HDFStore.select` modifies the passed columns list ([GH7212](#))
- Bug in `Categorical` repr with `display.width` of `None` in Python 3 ([GH10087](#))
- Bug in `to_json` with certain `orients` and a `CategoricalIndex` would segfault ([GH10317](#))
- Bug where some of the nan funcs do not have consistent return dtypes ([GH10251](#))
- Bug in `DataFrame.quantile` on checking that a valid axis was passed ([GH9543](#))
- Bug in `groupby.apply` aggregation for `Categorical` not preserving categories ([GH10138](#))
- Bug in `to_csv` where `date_format` is ignored if the `datetime` is fractional ([GH10209](#))
- Bug in `DataFrame.to_json` with mixed data types ([GH10289](#))
- Bug in cache updating when consolidating ([GH10264](#))
- Bug in `mean()` where integer dtypes can overflow ([GH10172](#))
- Bug where `Panel.from_dict` does not set dtype when specified ([GH10058](#))
- Bug in `Index.union` raises `AttributeError` when passing array-likes. ([GH10149](#))
- Bug in `Timestamp`'s `microsecond`, `quarter`, `dayofyear`, `week` and `daysinmonth` properties return `np.int` type, not built-in `int`. ([GH10050](#))
- Bug in `NaT` raises `AttributeError` when accessing to `daysinmonth`, `dayofweek` properties. ([GH10096](#))
- Bug in `Index` repr when using the `max_seq_items=None` setting ([GH10182](#)).
- Bug in getting timezone data with `dateutil` on various platforms ( [GH9059](#), [GH8639](#), [GH9663](#), [GH10121](#))
- Bug in displaying datetimes with mixed frequencies; display 'ms' datetimes to the proper precision. ([GH10170](#))
- Bug in `setitem` where type promotion is applied to the entire block ([GH10280](#))
- Bug in `Series` arithmetic methods may incorrectly hold names ([GH10068](#))
- Bug in `GroupBy.get_group` when grouping on multiple keys, one of which is categorical. ([GH10132](#))
- Bug in `DatetimeIndex` and `TimedeltaIndex` names are lost after `timedelta` arithmetics ( [GH9926](#))
- Bug in `DataFrame` construction from nested dict with `datetime64` ([GH10160](#))
- Bug in `Series` construction from dict with `datetime64` keys ([GH9456](#))
- Bug in `Series.plot(label="LABEL")` not correctly setting the label ([GH10119](#))
- Bug in `plot` not defaulting to `matplotlib.axes.grid` setting ([GH9792](#))
- Bug causing strings containing an exponent, but no decimal to be parsed as `int` instead of `float` in `engine='python'` for the `read_csv` parser ([GH9565](#))
- Bug in `Series.align` resets name when `fill_value` is specified ([GH10067](#))
- Bug in `read_csv` causing index name not to be set on an empty `DataFrame` ([GH10184](#))
- Bug in `SparseSeries.abs` resets name ([GH10241](#))
- Bug in `TimedeltaIndex` slicing may reset freq ([GH10292](#))
- Bug in `GroupBy.get_group` raises `ValueError` when group key contains `NaT` ([GH6992](#))
- Bug in `SparseSeries` constructor ignores input data name ([GH10258](#))

- Bug in `Categorical.remove_categories` causing a `ValueError` when removing the `NaN` category if underlying dtype is floating-point ([GH10156](#))
- Bug where `infer_freq` infers timerule (WOM-5XXX) unsupported by `to_offset` ([GH9425](#))
- Bug in `DataFrame.to_hdf()` where table format would raise a seemingly unrelated error for invalid (non-string) column names. This is now explicitly forbidden. ([GH9057](#))
- Bug to handle masking empty `DataFrame` ([GH10126](#)).
- Bug where MySQL interface could not handle numeric table/column names ([GH10255](#))
- Bug in `read_csv` with a `date_parser` that returned a `datetime64` array of other time resolution than `[ns]` ([GH10245](#))
- Bug in `Panel.apply` when the result has `ndim=0` ([GH10332](#))
- Bug in `read_hdf` where `auto_close` could not be passed ([GH9327](#)).
- Bug in `read_hdf` where open stores could not be used ([GH10330](#)).
- Bug in adding empty `DataFrames`, now results in a `DataFrame` that `.equals` an empty `DataFrame` ([GH10181](#)).
- Bug in `to_hdf` and `HDFStore` which did not check that complib choices were valid ([GH4582](#), [GH8874](#)).

## 1.18 v0.16.1 (May 11, 2015)

This is a minor bug-fix release from 0.16.0 and includes a a large number of bug fixes along several new features, enhancements, and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Support for a `CategoricalIndex`, a category based index, see [here](#)
- New section on how-to-contribute to *pandas*, see [here](#)
- Revised “Merge, join, and concatenate” documentation, including graphical examples to make it easier to understand each operations, see [here](#)
- New method `sample` for drawing random samples from Series, DataFrames and Panels. See [here](#)
- The default `Index` printing has changed to a more uniform format, see [here](#)
- `BusinessHour` datetime-offset is now supported, see [here](#)
- Further enhancement to the `.str` accessor to make string operations easier, see [here](#)

### What’s new in v0.16.1

- *Enhancements*
  - *CategoricalIndex*
  - *Sample*
  - *String Methods Enhancements*
  - *Other Enhancements*
- *API changes*
  - *Deprecations*

- *Index Representation*
- *Performance Improvements*
- *Bug Fixes*

**Warning:** In pandas 0.17.0, the sub-package `pandas.io.data` will be removed in favor of a separately installable package ([GH8961](#)).

## 1.18.1 Enhancements

### 1.18.1.1 CategoricalIndex

We introduce a `CategoricalIndex`, a new type of index object that is useful for supporting indexing with duplicates. This is a container around a `Categorical` (introduced in v0.15.0) and allows efficient indexing and storage of an index with a large number of duplicated elements. Prior to 0.16.1, setting the index of a `DataFrame`/`Series` with a `category` dtype would convert this to regular object-based `Index`.

```
In [1]: df = DataFrame({'A' : np.arange(6),
...:                   'B' : Series(list('aabbca')).astype('category',
...:                                                     categories=list('cab'))
...:                   })
...:

In [2]: df
Out[2]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [3]: df.dtypes
Out[3]:
A      int64
B    category
dtype: object

In [4]: df.B.cat.categories
Out[4]: Index(['c', 'a', 'b'], dtype='object')
```

setting the index, will create a `CategoricalIndex`

```
In [5]: df2 = df.set_index('B')

In [6]: df2.index
Out[6]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],
↳ ordered=False, name='B', dtype='category')
```

indexing with `__getitem__/.iloc/.loc/.ix` works similarly to an `Index` with duplicates. The indexers **MUST** be in the category or the operation will raise.

```
In [7]: df2.loc['a']
Out[7]:
A
B
a    0
a    1
a    5
```

and preserves the CategoricalIndex

```
In [8]: df2.loc['a'].index
Out[8]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,
↳name='B', dtype='category')
```

sorting will order by the order of the categories

```
In [9]: df2.sort_index()
Out[9]:
A
B
c    4
a    0
a    1
a    5
b    2
b    3
```

groupby operations on the index will preserve the index nature as well

```
In [10]: df2.groupby(level=0).sum()
Out[10]:
A
B
c    4
a    6
b    5

In [11]: df2.groupby(level=0).sum().index
Out[11]: CategoricalIndex(['c', 'a', 'b'], categories=['c', 'a', 'b'], ordered=False,
↳name='B', dtype='category')
```

reindexing operations, will return a resulting index based on the type of the passed indexer, meaning that passing a list will return a plain-old-Index; indexing with a Categorical will return a CategoricalIndex, indexed according to the categories of the PASSED Categorical dtype. This allows one to arbitrarily index these even with values NOT in the categories, similarly to how you can reindex ANY pandas index.

```
In [12]: df2.reindex(['a', 'e'])
Out[12]:
A
B
a    0.0
a    1.0
a    5.0
e    NaN

In [13]: df2.reindex(['a', 'e']).index
Out[13]: Index(['a', 'a', 'a', 'e'], dtype='object', name='B')
```

(continues on next page)

(continued from previous page)

```

In [14]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde'))))
Out[14]:
      A
B
a    0.0
a    1.0
a    5.0
e    NaN

In [15]: df2.reindex(pd.Categorical(['a','e'],categories=list('abcde'))).index
Out[15]: CategoricalIndex(['a', 'a', 'a', 'e'], categories=['a', 'b', 'c', 'd', 'e'],
↳ordered=False, name='B', dtype='category')

```

See the [documentation](#) for more. ([GH7629](#), [GH10038](#), [GH10039](#))

### 1.18.1.2 Sample

Series, DataFrames, and Panels now have a new method: `sample()`. The method accepts a specific number of rows or columns to return, or a fraction of the total number of rows or columns. It also has options for sampling with or without replacement, for passing in a column for weights for non-uniform sampling, and for setting seed values to facilitate replication. ([GH2419](#))

```

In [1]: example_series = Series([0,1,2,3,4,5])

# When no arguments are passed, returns 1
In [2]: example_series.sample()
Out[2]:
3      3
dtype: int64

# One may specify either a number of rows:
In [3]: example_series.sample(n=3)
Out[3]:
5      5
1      1
4      4
dtype: int64

# Or a fraction of the rows:
In [4]: example_series.sample(frac=0.5)
Out[4]:
4      4
1      1
0      0
dtype: int64

# weights are accepted.
In [5]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [6]: example_series.sample(n=3, weights=example_weights)
Out[6]:
2      2
3      3
5      5

```

(continues on next page)

(continued from previous page)

```
dtype: int64

# weights will also be normalized if they do not sum to one,
# and missing values will be treated as zeros.
In [7]: example_weights2 = [0.5, 0, 0, 0, None, np.nan]

In [8]: example_series.sample(n=1, weights=example_weights2)
Out[8]:
0      0
dtype: int64
```

When applied to a DataFrame, one may pass the name of a column to specify sampling weights when sampling from rows.

```
In [9]: df = DataFrame({'col1':[9,8,7,6], 'weight_column':[0.5, 0.4, 0.1, 0]})

In [10]: df.sample(n=3, weights='weight_column')
Out[10]:
   col1  weight_column
0     9             0.5
1     8             0.4
2     7             0.1
```

### 1.18.1.3 String Methods Enhancements

*Continuing from v0.16.0*, the following enhancements make string operations easier and more consistent with standard python string operations.

- Added StringMethods (`.str` accessor) to Index ([GH9068](#))

The `.str` accessor is now available for both Series and Index.

```
In [11]: idx = Index([' jack', 'jill ', ' jesse ', 'frank'])

In [12]: idx.str.strip()
Out[12]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')
```

One special case for the `.str` accessor on Index is that if a string method returns bool, the `.str` accessor will return a `np.array` instead of a boolean Index ([GH8875](#)). This enables the following expression to work naturally:

```
In [13]: idx = Index(['a1', 'a2', 'b1', 'b2'])

In [14]: s = Series(range(4), index=idx)

In [15]: s
Out[15]:
a1    0
a2    1
b1    2
b2    3
dtype: int64

In [16]: idx.str.startswith('a')
Out[16]: array([ True,  True, False, False], dtype=bool)
```

(continues on next page)



```
In [17]: s[s.index.str.startswith('a')]
\\
a1      0
a2      1
dtype: int64
```

```

In [24]: from pandas.tseries.offsets import BusinessHour

In [25]: Timestamp('2014-08-01 09:00') + BusinessHour()
Out[25]: Timestamp('2014-08-01 10:00:00')

In [26]: Timestamp('2014-08-01 07:00') + BusinessHour()
Out[26]: Timestamp('2014-08-01 10:00:00')

In [27]: Timestamp('2014-08-01 16:30') + BusinessHour()
Out[27]: Timestamp('2014-08-04 09:30:00')

```

- `DataFrame.diff` now takes an `axis` parameter that determines the direction of differencing ([GH9727](#))
- Allow `clip`, `clip_lower`, and `clip_upper` to accept array-like arguments as thresholds (This is a regression from 0.11.0). These methods now have an `axis` parameter which determines how the Series or DataFrame will be aligned with the threshold(s). ([GH6966](#))
- `DataFrame.mask()` and `Series.mask()` now support same keywords as `where` ([GH8801](#))
- `drop` function can now accept `errors` keyword to suppress `ValueError` raised when any of label does not exist in the target data. ([GH6736](#))

```

In [28]: df = DataFrame(np.random.randn(3, 3), columns=['A', 'B', 'C'])

In [29]: df.drop(['A', 'X'], axis=1, errors='ignore')
Out[29]:
      B      C
0  1.058969 -0.397840
1  1.047579  1.045938
2 -0.122092  0.124713

```

- Add support for separating years and quarters using dashes, for example 2014-Q1. ([GH9688](#))
- Allow conversion of values with dtype `datetime64` or `timedelta64` to strings using `astype(str)` ([GH9757](#))
- `get_dummies` function now accepts `sparse` keyword. If set to `True`, the return DataFrame is sparse, e.g. `SparseDataFrame`. ([GH8823](#))
- `Period` now accepts `datetime64` as value input. ([GH9054](#))
- Allow `timedelta` string conversion when leading zero is missing from time definition, ie `0:00:00` vs `00:00:00`. ([GH9570](#))
- Allow `Panel.shift` with `axis='items'` ([GH9890](#))
- Trying to write an excel file now raises `NotImplementedError` if the DataFrame has a `MultiIndex` instead of writing a broken Excel file. ([GH9794](#))
- Allow `Categorical.add_categories` to accept `Series` or `np.array`. ([GH9927](#))
- Add/delete `str/dt/cat` accessors dynamically from `__dir__`. ([GH9910](#))
- Add `normalize` as a `dt` accessor method. ([GH10047](#))
- `DataFrame` and `Series` now have `_constructor_expanddim` property as overridable constructor for one higher dimensionality data. This should be used only when it is really needed, see [here](#)
- `pd.lib.infer_dtype` now returns `'bytes'` in Python 3 where appropriate. ([GH10032](#))

## 1.18.2 API changes

- When passing in an `ax` to `df.plot(..., ax=ax)`, the `sharex` kwarg will now default to `False`. The result is that the visibility of `xlabels` and `xticklabels` will not anymore be changed. You have to do that by yourself for the right axes in your figure or set `sharex=True` explicitly (but this changes the visible for all axes in the figure, not only the one which is passed in!). If pandas creates the subplots itself (e.g. no passed in `ax` kwarg), then the default is still `sharex=True` and the visibility changes are applied.
- `assign()` now inserts new columns in alphabetical order. Previously the order was arbitrary. (GH9777)
- By default, `read_csv` and `read_table` will now try to infer the compression type based on the file extension. Set `compression=None` to restore the previous behavior (no decompression). (GH9770)

### 1.18.2.1 Deprecations

- `Series.str.split`'s `return_type` keyword was removed in favor of `expand` (GH9847)

## 1.18.3 Index Representation

The string representation of `Index` and its sub-classes have now been unified. These will show a single-line display if there are few values; a wrapped multi-line display for a lot of values (but less than `display.max_seq_items`; if lots of items (`> display.max_seq_items`) will show a truncated display (the head and tail of the data). The formatting for `MultiIndex` is unchanged (a multi-line wrapped display). The display width responds to the option `display.max_seq_items`, which is defaulted to 100. (GH6482)

Previous Behavior

```
In [2]: pd.Index(range(4), name='foo')
Out[2]: Int64Index([0, 1, 2, 3], dtype='int64')

In [3]: pd.Index(range(104), name='foo')
Out[3]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
↳19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
↳40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
↳61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
↳82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, ...], dtype=
↳'int64')

In [4]: pd.date_range('20130101', periods=4, name='foo', tz='US/Eastern')
Out[4]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00-05:00, ..., 2013-01-04 00:00:00-05:00]
Length: 4, Freq: D, Timezone: US/Eastern

In [5]: pd.date_range('20130101', periods=104, name='foo', tz='US/Eastern')
Out[5]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2013-01-01 00:00:00-05:00, ..., 2013-04-14 00:00:00-04:00]
Length: 104, Freq: D, Timezone: US/Eastern
```

New Behavior

```
In [30]: pd.set_option('display.width', 80)

In [31]: pd.Index(range(4), name='foo')
Out[31]: RangeIndex(start=0, stop=4, step=1, name='foo')
```

(continues on next page)

(continues on next page)

(continued from previous page)

```
DatetimeIndex(['2013-01-01 00:00:00-05:00', '2013-01-02 00:00:00-05:00',
              '2013-01-03 00:00:00-05:00', '2013-01-04 00:00:00-05:00',
              '2013-01-05 00:00:00-05:00', '2013-01-06 00:00:00-05:00',
              '2013-01-07 00:00:00-05:00', '2013-01-08 00:00:00-05:00',
              '2013-01-09 00:00:00-05:00', '2013-01-10 00:00:00-05:00',
              ...,
              '2013-04-05 00:00:00-04:00', '2013-04-06 00:00:00-04:00',
              '2013-04-07 00:00:00-04:00', '2013-04-08 00:00:00-04:00',
              '2013-04-09 00:00:00-04:00', '2013-04-10 00:00:00-04:00',
              '2013-04-11 00:00:00-04:00', '2013-04-12 00:00:00-04:00',
              '2013-04-13 00:00:00-04:00', '2013-04-14 00:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', name='foo', length=104, freq='D')
```

### 1.18.4 Performance Improvements

- Improved csv write performance with mixed dtypes, including datetimes by up to 5x ([GH9940](#))
- Improved csv write performance generally by 2x ([GH9940](#))
- Improved the performance of `pd.lib.max_len_string_array` by 5-7x ([GH10024](#))

### 1.18.5 Bug Fixes

- Bug where labels did not appear properly in the legend of `DataFrame.plot()`, passing `label=` arguments works, and Series indices are no longer mutated. ([GH9542](#))
- Bug in json serialization causing a segfault when a frame had zero length. ([GH9805](#))
- Bug in `read_csv` where missing trailing delimiters would cause segfault. ([GH5664](#))
- Bug in retaining index name on appending ([GH9862](#))
- Bug in `scatter_matrix` draws unexpected axis ticklabels ([GH5662](#))
- Fixed bug in `StataWriter` resulting in changes to input `DataFrame` upon save ([GH9795](#)).
- Bug in `transform` causing length mismatch when null entries were present and a fast aggregator was being used ([GH9697](#))
- Bug in `equals` causing false negatives when block order differed ([GH9330](#))
- Bug in grouping with multiple `pd.Grouper` where one is non-time based ([GH10063](#))
- Bug in `read_sql_table` error when reading postgres table with timezone ([GH7139](#))
- Bug in `DataFrame` slicing may not retain metadata ([GH9776](#))
- Bug where `TimedeltaIndex` were not properly serialized in fixed `HDFStore` ([GH9635](#))
- Bug with `TimedeltaIndex` constructor ignoring name when given another `TimedeltaIndex` as data ([GH10025](#)).
- Bug in `DataFrameFormatter._get_formatted_index` with not applying `max_colwidth` to the `DataFrame` index ([GH7856](#))
- Bug in `.loc` with a read-only ndarray data source ([GH10043](#))
- Bug in `groupby.apply()` that would raise if a passed user defined function either returned only `None` (for all input). ([GH9685](#))

- Always use temporary files in pytables tests ([GH9992](#))
- Bug in plotting continuously using `secondary_y` may not show legend properly. ([GH9610](#), [GH9779](#))
- Bug in `DataFrame.plot(kind="hist")` results in `TypeError` when `DataFrame` contains non-numeric columns ([GH9853](#))
- Bug where repeated plotting of `DataFrame` with a `DatetimeIndex` may raise `TypeError` ([GH9852](#))
- Bug in `setup.py` that would allow an incompat cython version to build ([GH9827](#))
- Bug in plotting `secondary_y` incorrectly attaches `right_ax` property to secondary axes specifying itself recursively. ([GH9861](#))
- Bug in `Series.quantile` on empty `Series` of type `Datetime` or `Timedelta` ([GH9675](#))
- Bug in `where` causing incorrect results when upcasting was required ([GH9731](#))
- Bug in `FloatArrayFormatter` where decision boundary for displaying “small” floats in decimal format is off by one order of magnitude for a given `display.precision` ([GH9764](#))
- Fixed bug where `DataFrame.plot()` raised an error when both `color` and `style` keywords were passed and there was no color symbol in the style strings ([GH9671](#))
- Not showing a `DeprecationWarning` on combining list-likes with an `Index` ([GH10083](#))
- Bug in `read_csv` and `read_table` when using `skip_rows` parameter if blank lines are present. ([GH9832](#))
- Bug in `read_csv()` interprets `index_col=True` as 1 ([GH9798](#))
- Bug in index equality comparisons using `==` failing on `Index/MultiIndex` type incompatibility ([GH9785](#))
- Bug in which `SparseDataFrame` could not take `nan` as a column name ([GH8822](#))
- Bug in `to_msgpack` and `read_msgpack` `zlib` and `blosc` compression support ([GH9783](#))
- Bug `GroupBy.size` doesn’t attach index name properly if grouped by `TimeGrouper` ([GH9925](#))
- Bug causing an exception in slice assignments because `length_of_indexer` returns wrong results ([GH9995](#))
- Bug in `csv` parser causing lines with initial whitespace plus one non-space character to be skipped. ([GH9710](#))
- Bug in `C` `csv` parser causing spurious NaNs when data started with newline followed by whitespace. ([GH10022](#))
- Bug causing elements with a null group to spill into the final group when grouping by a `Categorical` ([GH9603](#))
- Bug where `.iloc` and `.loc` behavior is not consistent on empty dataframes ([GH9964](#))
- Bug in invalid attribute access on a `TimedeltaIndex` incorrectly raised `ValueError` instead of `AttributeError` ([GH9680](#))
- Bug in unequal comparisons between categorical data and a scalar, which was not in the categories (e.g. `Series(Categorical(list("abc")), ordered=True) > "d"`. This returned `False` for all elements, but now raises a `TypeError`. Equality comparisons also now return `False` for `==` and `True` for `!=`. ([GH9848](#))
- Bug in `DataFrame.__setitem__` when right hand side is a dictionary ([GH9874](#))
- Bug in `where` when `dtype` is `datetime64/timedelta64`, but `dtype` of other is not ([GH9804](#))
- Bug in `MultiIndex.sortlevel()` results in unicode level name breaks ([GH9856](#))
- Bug in which `groupby.transform` incorrectly enforced output dtypes to match input dtypes. ([GH9807](#))
- Bug in `DataFrame` constructor when `columns` parameter is set, and `data` is an empty list ([GH9939](#))

- Bug in bar plot with `log=True` raises `TypeError` if all values are less than 1 (GH9905)
- Bug in horizontal bar plot ignores `log=True` (GH9905)
- Bug in PyTables queries that did not return proper results using the index (GH8265, GH9676)
- Bug where dividing a dataframe containing values of type `Decimal` by another `Decimal` would raise. (GH9787)
- Bug where using DataFrames `asfreq` would remove the name of the index. (GH9885)
- Bug causing extra index point when `resample` BM/BQ (GH9756)
- Changed caching in `AbstractHolidayCalendar` to be at the instance level rather than at the class level as the latter can result in unexpected behaviour. (GH9552)
- Fixed latex output for multi-indexed dataframes (GH9778)
- Bug causing an exception when setting an empty range using `DataFrame.loc` (GH9596)
- Bug in hiding ticklabels with subplots and shared axes when adding a new plot to an existing grid of axes (GH9158)
- Bug in `transform` and `filter` when grouping on a categorical variable (GH9921)
- Bug in `transform` when groups are equal in number and dtype to the input index (GH9700)
- Google BigQuery connector now imports dependencies on a per-method basis. (GH9713)
- Updated BigQuery connector to no longer use deprecated `oauth2client.tools.run()` (GH8327)
- Bug in subclassed `DataFrame`. It may not return the correct class, when slicing or subsetting it. (GH9632)
- Bug in `.median()` where non-float null values are not handled correctly (GH10040)
- Bug in `Series.fillna()` where it raises if a numerically convertible string is given (GH10092)

## 1.19 v0.16.0 (March 22, 2015)

This is a major release from 0.15.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- `DataFrame.assign` method, see [here](#)
- `Series.to_coo/from_coo` methods to interact with `scipy.sparse`, see [here](#)
- Backwards incompatible change to `Timedelta` to conform the `.seconds` attribute with `datetime.timedelta`, see [here](#)
- Changes to the `.loc` slicing API to conform with the behavior of `.ix` see [here](#)
- Changes to the default for ordering in the `Categorical` constructor, see [here](#)
- Enhancement to the `.str` accessor to make string operations easier, see [here](#)
- The `pandas.tools.rplot`, `pandas.sandbox.qtpandas` and `pandas.rpy` modules are deprecated. We refer users to external packages like [seaborn](#), [pandas-qt](#) and [rpy2](#) for similar or equivalent functionality, see [here](#)

Check the [API Changes](#) and [deprecations](#) before updating.

**What's new in v0.16.0**

- *New features*
  - *DataFrame Assign*
  - *Interaction with scipy.sparse*
  - *String Methods Enhancements*
  - *Other enhancements*
- *Backwards incompatible API changes*
  - *Changes in Timedelta*
  - *Indexing Changes*
  - *Categorical Changes*
  - *Other API Changes*
  - *Deprecations*
  - *Removal of prior version deprecations/changes*
- *Performance Improvements*
- *Bug Fixes*

**1.19.1 New features****1.19.1.1 DataFrame Assign**

Inspired by `dplyr`'s `mutate` verb, `DataFrame` has a new `assign()` method. The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or `NumPy` array), or a function of one argument to be called on the `DataFrame`. The new values are inserted, and the entire `DataFrame` (with all original and new columns) is returned.

```
In [1]: iris = read_csv('data/iris.data')
```

```
In [2]: iris.head()
```

```
Out[2]:
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

```
In [3]: iris.assign(sepal_ratio=iris['SepalWidth'] / iris['SepalLength']).head()
```

```

////////////////////////////////////
↪

```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name	sepal_ratio
0	5.1	3.5	1.4	0.2	Iris-setosa	0.686275
1	4.9	3.0	1.4	0.2	Iris-setosa	0.612245
2	4.7	3.2	1.3	0.2	Iris-setosa	0.680851
3	4.6	3.1	1.5	0.2	Iris-setosa	0.673913
4	5.0	3.6	1.4	0.2	Iris-setosa	0.720000



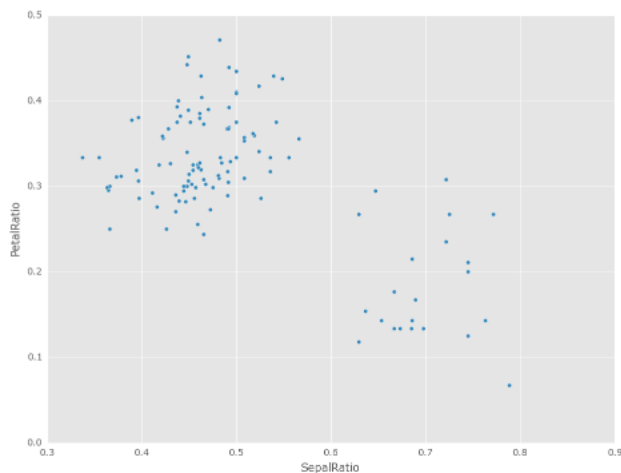
Above was an example of inserting a precomputed value. We can also pass in a function to be evaluated.

```
In [4]: iris.assign(sepal_ratio = lambda x: (x['SepalWidth'] /
...:                                         x['SepalLength'])).head()
Out[4]:
```

	SepalLength	SepalWidth	PetalLength	PetalWidth	Name	sepal_ratio
0	5.1	3.5	1.4	0.2	Iris-setosa	0.686275
1	4.9	3.0	1.4	0.2	Iris-setosa	0.612245
2	4.7	3.2	1.3	0.2	Iris-setosa	0.680851
3	4.6	3.1	1.5	0.2	Iris-setosa	0.673913
4	5.0	3.6	1.4	0.2	Iris-setosa	0.720000

The power of `assign` comes when used in chains of operations. For example, we can limit the DataFrame to just those with a Sepal Length greater than 5, calculate the ratio, and plot

```
In [5]: (iris.query('SepalLength > 5')
...:      .assign(SepalRatio = lambda x: x.SepalWidth / x.SepalLength,
...:              PetalRatio = lambda x: x.PetalWidth / x.PetalLength)
...:      .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
Out[5]: <matplotlib.axes._subplots.AxesSubplot at 0x1c41df41d0>
```



See the [documentation](#) for more. (GH9229)

### 1.19.1.2 Interaction with `scipy.sparse`

Added `SparseSeries.to_coo()` and `SparseSeries.from_coo()` methods (GH8048) for converting to and from `scipy.sparse.coo_matrix` instances (see [here](#)). For example, given a `SparseSeries` with `MultiIndex` we can convert to a `scipy.sparse.coo_matrix` by specifying the row and column labels as index levels:

```
In [6]: from numpy import nan

In [7]: s = Series([3.0, nan, 1.0, 3.0, nan, nan])

In [8]: s.index = MultiIndex.from_tuples([(1, 2, 'a', 0),
...:                                     (1, 2, 'a', 1),
...:                                     (1, 1, 'b', 0),
...:                                     (1, 1, 'b', 1),
...:                                     (2, 1, 'b', 0),
```

(continues on next page)

(continued from previous page)

[illegible]

The `from_coo` method is a convenience method for creating a `SparseSeries` from a `scipy.sparse.coo_matrix`:

[illegible]

### 1.19.1.3 String Methods Enhancements

- Following new methods are accessible via `.str` accessor to apply the function to each values. This is intended to make it more consistent with standard methods on strings. ([GH9282](#), [GH9352](#), [GH9386](#), [GH9387](#), [GH9439](#))

		Methods		
isalnum()	isalpha()	isdigit()	isdigit()	isspace()
islower()	isupper()	istitle()	isnumeric()	isdecimal()
find()	rfind()	ljust()	rjust()	zfill()

```
In [23]: s = Series(['abcd', '3456', 'EFGH'])

In [24]: s.str.isalpha()
Out[24]:
0      True
1     False
2      True
dtype: bool

In [25]: s.str.find('ab')
Out[25]:
0      0
```

(continues on next page)

(continued from previous page)

```
1    -1
2    -1
dtype: int64
```

- `Series.str.pad()` and `Series.str.center()` now accept `fillchar` option to specify filling character (GH9352)

```
In [26]: s = Series(['12', '300', '25'])

In [27]: s.str.pad(5, fillchar='_')
Out[27]:
0    ____12
1    __300
2    ____25
dtype: object
```

- Added `Series.str.slice_replace()`, which previously raised `NotImplementedError` (GH8888)

```
In [28]: s = Series(['ABCD', 'EFGH', 'IJK'])

In [29]: s.str.slice_replace(1, 3, 'X')
Out[29]:
0    AXD
1    EXH
2     IX
dtype: object

# replaced with empty char
In [30]: s.str.slice_replace(0, 1)
Out[30]:
0    BCD
1    FGH
2     JK
dtype: object
```

### 1.19.1.4 Other enhancements

- Reindex now supports `method='nearest'` for frames or series with a monotonic increasing or decreasing index (GH9258):

```
In [31]: df = pd.DataFrame({'x': range(5)})

In [32]: df.reindex([0.2, 1.8, 3.5], method='nearest')
Out[32]:
      x
0.2  0
1.8  2
3.5  4
```

This method is also exposed by the lower level `Index.get_indexer` and `Index.get_loc` methods.

- The `read_excel()` function's `sheetname` argument now accepts a list and `None`, to get multiple or all sheets respectively. If more than one sheet is specified, a dictionary is returned. (GH9450)

```
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
pd.read_excel('path_to_file.xls', sheetname=['Sheet1', 3])
```

- Allow Stata files to be read incrementally with an iterator; support for long strings in Stata files. See the docs [here](#) (GH9493:).
- Paths beginning with ~ will now be expanded to begin with the user's home directory (GH9066)
- Added time interval selection in `get_data_yahoo` (GH9071)
- Added `Timestamp.to_datetime64()` to complement `Timedelta.to_timedelta64()` (GH9255)
- `tsseries.frequencies.to_offset()` now accepts `Timedelta` as input (GH9064)
- Lag parameter was added to the autocorrelation method of `Series`, defaults to lag-1 autocorrelation (GH9192)
- `Timedelta` will now accept `nanoseconds` keyword in constructor (GH9273)
- SQL code now safely escapes table and column names (GH8986)
- Added auto-complete for `Series.str.<tab>`, `Series.dt.<tab>` and `Series.cat.<tab>` (GH9322)
- `Index.get_indexer` now supports `method='pad'` and `method='backfill'` even for any target array, not just monotonic targets. These methods also work for monotonic decreasing as well as monotonic increasing indexes (GH9258).
- `Index.asof` now works on all index types (GH9258).
- A `verbose` argument has been augmented in `io.read_excel()`, defaults to `False`. Set to `True` to print sheet names as they are parsed. (GH9450)
- Added `days_in_month` (compatibility alias `daysinmonth`) property to `Timestamp`, `DatetimeIndex`, `Period`, `PeriodIndex`, and `Series.dt` (GH9572)
- Added `decimal` option in `to_csv` to provide formatting for non-',' decimal separators (GH781)
- Added `normalize` option for `Timestamp` to normalized to midnight (GH8794)
- Added example for `DataFrame` import to R using HDF5 file and `rhdf5` library. See the [documentation](#) for more (GH9636).

## 1.19.2 Backwards incompatible API changes

### 1.19.2.1 Changes in Timedelta

In v0.15.0 a new scalar type `Timedelta` was introduced, that is a sub-class of `datetime.timedelta`. Mentioned [here](#) was a notice of an API change w.r.t. the `.seconds` accessor. The intent was to provide a user-friendly set of accessors that give the 'natural' value for that unit, e.g. if you had a `Timedelta('1 day, 10:11:12')`, then `.seconds` would return 12. However, this is at odds with the definition of `datetime.timedelta`, which defines `.seconds` as `10 * 3600 + 11 * 60 + 12 == 36672`.

So in v0.16.0, we are restoring the API to match that of `datetime.timedelta`. Further, the component values are still available through the `.components` accessor. This affects the `.seconds` and `.microseconds` accessors, and removes the `.hours`, `.minutes`, `.milliseconds` accessors. These changes affect `TimedeltaIndex` and the `Series .dt` accessor as well. (GH9185, GH9139)

Previous Behavior

```
In [2]: t = pd.Timedelta('1 day, 10:11:12.100123')
In [3]: t.days
Out[3]: 1
```

(continues on next page)

(continued from previous page)

```
In [4]: t.seconds
Out[4]: 12

In [5]: t.microseconds
Out[5]: 123
```

### New Behavior

```
In [33]: t = pd.Timedelta('1 day, 10:11:12.100123')

In [34]: t.days
Out[34]: 1

In [35]: t.seconds
Out[35]: 36672

In [36]: t.microseconds
Out[36]: 100123
```

Using `.components` allows the full component access

```
In [37]: t.components
Out[37]: Components(days=1, hours=10, minutes=11, seconds=12, milliseconds=100,
↳ microseconds=123, nanoseconds=0)

In [38]: t.components.seconds
Out[38]: 12
```

### 1.19.2.2 Indexing Changes

The behavior of a small sub-set of edge cases for using `.loc` have changed ([GH8613](#)). Furthermore we have improved the content of the error messages that are raised:

- Slicing with `.loc` where the start and/or stop bound is not found in the index is now allowed; this previously would raise a `KeyError`. This makes the behavior the same as `.ix` in this case. This change is only for slicing, not when indexing with a single label.

```
In [39]: df = DataFrame(np.random.randn(5,4),
.....:                  columns=list('ABCD'),
.....:                  index=date_range('20130101', periods=5))
.....:

In [40]: df
Out[40]:
```

	A	B	C	D
2013-01-01	-0.322795	0.841675	2.390961	0.076200
2013-01-02	-0.566446	0.036142	-2.074978	0.247792
2013-01-03	-0.897157	-0.136795	0.018289	0.755414
2013-01-04	0.215269	0.841009	-1.445810	-1.401973
2013-01-05	-0.100918	-0.548242	-0.144620	0.354020

```
In [41]: s = Series(range(5), [-2, -1, 1, 2, 3])

In [42]: s
```

(continues on next page)

(continued from previous page)

```
Out[42]:
-2      0
-1      1
 1      2
 2      3
 3      4
dtype: int64
```

## Previous Behavior

```
In [4]: df.loc['2013-01-02':'2013-01-10']
KeyError: 'stop bound [2013-01-10] is not in the [index]'
```

```
In [6]: s.loc[-10:3]
KeyError: 'start bound [-10] is not the [index]'
```

### New Behavior

```
In [43]: df.loc['2013-01-02':'2013-01-10']
Out[43]:
```

	A	B	C	D
2013-01-02	-0.566446	0.036142	-2.074978	0.247792
2013-01-03	-0.897157	-0.136795	0.018289	0.755414
2013-01-04	0.215269	0.841009	-1.445810	-1.401973
2013-01-05	-0.100918	-0.548242	-0.144620	0.354020

```
In [44]: s.loc[-10:3]
//////////
↵
-2      0
-1      1
1       2
2       3
3       4
dtype: int64
```

- Allow slicing with float-like values on an integer index for `.ix`. Previously this was only enabled for `.loc`:

## Previous Behavior

```
In [8]: s.ix[-1.0:2]
TypeError: the slice start value [-1.0] is not a proper indexer for this index_
↪type (Int64Index)
```

## New Behavior

```
In [2]: s.ix[-1.0:2]
Out[2]:
-1      1
 1      2
 2      3
dtype: int64
```

- Provide a useful exception for indexing with an invalid type for that index when using `.loc`. For example trying to use `.loc` on an index of type `DatetimeIndex` or `PeriodIndex` or `TimedeltaIndex`, with an integer (or a float).

## Previous Behavior

```
In [4]: df.loc[2:3]
KeyError: 'start bound [2] is not the [index]'
```

#### New Behavior

```
In [4]: df.loc[2:3]
TypeError: Cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'
↳> with <type 'int'> keys
```

### 1.19.2.3 Categorical Changes

In prior versions, `Categoricals` that had an unspecified ordering (meaning no `ordered` keyword was passed) were defaulted as ordered `Categoricals`. Going forward, the `ordered` keyword in the `Categorical` constructor will default to `False`. Ordering must now be explicit.

Furthermore, previously you *could* change the `ordered` attribute of a `Categorical` by just setting the attribute, e.g. `cat.ordered=True`; This is now deprecated and you should use `cat.as_ordered()` or `cat.as_unordered()`. These will by default return a **new** object and not modify the existing object. (GH9347, GH9190)

#### Previous Behavior

```
In [3]: s = Series([0,1,2], dtype='category')

In [4]: s
Out[4]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [5]: s.cat.ordered
Out[5]: True

In [6]: s.cat.ordered = False

In [7]: s
Out[7]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0, 1, 2]
```

#### New Behavior

```
In [45]: s = Series([0,1,2], dtype='category')

In [46]: s
Out[46]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0, 1, 2]
```

(continues on next page)



(continued from previous page)

```

In [47]: s.cat.ordered
\\Out [47]:
↪False

In [48]: s = s.cat.as_ordered()

In [49]: s
Out[49]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [50]: s.cat.ordered
\\Out [50]:
↪True

# you can set in the constructor of the Categorical
In [51]: s = Series(Categorical([0,1,2],ordered=True))

In [52]: s
Out[52]:
0    0
1    1
2    2
dtype: category
Categories (3, int64): [0 < 1 < 2]

In [53]: s.cat.ordered
\\Out [53]:
↪True

```

For ease of creation of series of categorical data, we have added the ability to pass keywords when calling `.astype()`. These are passed directly to the constructor.

```

In [54]: s = Series(["a", "b", "c", "a"]).astype('category', ordered=True)

In [55]: s
Out[55]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a < b < c]

In [56]: s = Series(["a", "b", "c", "a"]).astype('category', categories=list('abcdef'),
↪ordered=False)

In [57]: s
Out[57]:
0    a
1    b
2    c

```

(continues on next page)

(continued from previous page)

```
3      a
dtype: category
Categories (6, object): [a, b, c, d, e, f]
```

### 1.19.2.4 Other API Changes

- `Index.duplicated` now returns `np.array(dtype=bool)` rather than `Index(dtype=object)` containing `bool` values. (GH8875)
- `DataFrame.to_json` now returns accurate type serialisation for each column for frames of mixed dtype (GH9037)

Previously data was coerced to a common dtype before serialisation, which for example resulted in integers being serialised to floats:

```
In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
Out [2]: '{"f":{"0":3.0,"1":4.2},"i":{"0":1.0,"1":2.0}}'
```

Now each column is serialised using its correct dtype:

```
In [2]: pd.DataFrame({'i': [1,2], 'f': [3.0, 4.2]}).to_json()
Out [2]: '{"f":{"0":3.0,"1":4.2},"i":{"0":1,"1":2}}'
```

- `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex.summary` now output the same format. (GH9116)
- `TimedeltaIndex.freqstr` now output the same string format as `DatetimeIndex`. (GH9116)
- Bar and horizontal bar plots no longer add a dashed line along the info axis. The prior style can be achieved with matplotlib's `axhline` or `axvline` methods (GH9088).
- Series accessors `.dt`, `.cat` and `.str` now raise `AttributeError` instead of `TypeError` if the series does not contain the appropriate type of data (GH9617). This follows Python's built-in exception hierarchy more closely and ensures that tests like `hasattr(s, 'cat')` are consistent on both Python 2 and 3.
- Series now supports bitwise operation for integral types (GH9016). Previously even if the input dtypes were integral, the output dtype was coerced to `bool`.

Previous Behavior

```
In [2]: pd.Series([0,1,2,3], list('abcd')) | pd.Series([4,4,4,4], list('abcd'))
Out [2]:
a      True
b      True
c      True
d      True
dtype: bool
```

New Behavior. If the input dtypes are integral, the output dtype is also integral and the output values are the result of the bitwise operation.

```
In [2]: pd.Series([0,1,2,3], list('abcd')) | pd.Series([4,4,4,4], list('abcd'))
Out [2]:
a      4
b      5
c      6
```

(continues on next page)

```
d      7
dtype: int64
```

- ## Previous Behavior

```
0      inf
1      inf
dtype: float64
```

```
0      NaN
1      inf
dtype: float64
```

- Old behavior:

```
Out[4]: Timestamp('2000-01-31 00:00:00')
```

```
Out[57]: Timestamp('2000-02-28 00:00:00')
```

To reproduce the old behavior, simply add more precision to the label (e.g., use 2000-02-01 instead of 2000-02).

### 1.19.2.5 Deprecations

- The `rplot` trellis plotting interface is deprecated and will be removed in a future version. We refer to external packages like `seaborn` for similar but more refined functionality ([GH3445](#)). The documentation includes some examples how to convert your existing code using `rplot` to `seaborn`: [rplot docs](#).
- The `pandas.sandbox.qtpandas` interface is deprecated and will be removed in a future version. We refer users to the external package `pandas-qt`. ([GH9615](#))
- The `pandas.rpy` interface is deprecated and will be removed in a future version. Similar functionality can be accessed thru the `rpy2` project ([GH9602](#))
- Adding `DatetimeIndex/PeriodIndex` to another `DatetimeIndex/PeriodIndex` is being deprecated as a set-operation. This will be changed to a `TypeError` in a future version. `.union()` should be used for the union set operation. ([GH9094](#))
- Subtracting `DatetimeIndex/PeriodIndex` from another `DatetimeIndex/PeriodIndex` is being deprecated as a set-operation. This will be changed to an actual numeric subtraction yielding a `TimeDeltaIndex` in a future version. `.difference()` should be used for the differencing set operation. ([GH9094](#))

### 1.19.2.6 Removal of prior version deprecations/changes

- `DataFrame.pivot_table` and `crosstab`'s `rows` and `cols` keyword arguments were removed in favor of `index` and `columns` ([GH6581](#))
- `DataFrame.to_excel` and `DataFrame.to_csv` `cols` keyword argument was removed in favor of `columns` ([GH6581](#))
- Removed `convert_dummies` in favor of `get_dummies` ([GH6581](#))
- Removed `value_range` in favor of `describe` ([GH6581](#))

## 1.19.3 Performance Improvements

- Fixed a performance regression for `.loc` indexing with an array or list-like ([GH9126](#)).
- `DataFrame.to_json` 30x performance improvement for mixed dtype frames. ([GH9037](#))
- Performance improvements in `MultiIndex.duplicated` by working with labels instead of values ([GH9125](#))
- Improved the speed of `nunique` by calling `unique` instead of `value_counts` ([GH9129](#), [GH7771](#))
- Performance improvement of up to 10x in `DataFrame.count` and `DataFrame.dropna` by taking advantage of homogeneous/heterogeneous dtypes appropriately ([GH9136](#))
- Performance improvement of up to 20x in `DataFrame.count` when using a `MultiIndex` and the `level` keyword argument ([GH9163](#))
- Performance and memory usage improvements in `merge` when key space exceeds `int64` bounds ([GH9151](#))
- Performance improvements in multi-key `groupby` ([GH9429](#))
- Performance improvements in `MultiIndex.sortlevel` ([GH9445](#))
- Performance and memory usage improvements in `DataFrame.duplicated` ([GH9398](#))

- Cythonized Period ([GH9440](#))
- Decreased memory usage on `to_hdf` ([GH9648](#))

### 1.19.4 Bug Fixes

- Changed `.to_html` to remove leading/trailing spaces in table body ([GH4987](#))
- Fixed issue using `read_csv` on s3 with Python 3 ([GH9452](#))
- Fixed compatibility issue in `DatetimeIndex` affecting architectures where `numpy.int_` defaults to `numpy.int32` ([GH8943](#))
- Bug in Panel indexing with an object-like ([GH9140](#))
- Bug in the returned `Series.dt.components` index was reset to the default index ([GH9247](#))
- Bug in `Categorical.__getitem__`/`__setitem__` with listlike input getting incorrect results from indexer coercion ([GH9469](#))
- Bug in partial setting with a `DatetimeIndex` ([GH9478](#))
- Bug in groupby for integer and `datetime64` columns when applying an aggregator that caused the value to be changed when the number was sufficiently large ([GH9311](#), [GH6620](#))
- Fixed bug in `to_sql` when mapping a `Timestamp` object column (datetime column with timezone info) to the appropriate sqlalchemy type ([GH9085](#)).
- Fixed bug in `to_sql` `dtype` argument not accepting an instantiated `SQLAlchemy` type ([GH9083](#)).
- Bug in `.loc` partial setting with a `np.datetime64` ([GH9516](#))
- Incorrect dtypes inferred on datetimelike looking `Series` & on `.xs` slices ([GH9477](#))
- Items in `Categorical.unique()` (and `s.unique()` if `s` is of dtype `category`) now appear in the order in which they are originally found, not in sorted order ([GH9331](#)). This is now consistent with the behavior for other dtypes in pandas.
- Fixed bug on big endian platforms which produced incorrect results in `StataReader` ([GH8688](#)).
- Bug in `MultiIndex.has_duplicates` when having many levels causes an indexer overflow ([GH9075](#), [GH5873](#))
- Bug in `pivot` and `unstack` where nan values would break index alignment ([GH4862](#), [GH7401](#), [GH7403](#), [GH7405](#), [GH7466](#), [GH9497](#))
- Bug in `left join` on multi-index with `sort=True` or null values ([GH9210](#)).
- Bug in `MultiIndex` where inserting new keys would fail ([GH9250](#)).
- Bug in groupby when key space exceeds `int64` bounds ([GH9096](#)).
- Bug in `unstack` with `TimedeltaIndex` or `DatetimeIndex` and nulls ([GH9491](#)).
- Bug in `rank` where comparing floats with tolerance will cause inconsistent behaviour ([GH8365](#)).
- Fixed character encoding bug in `read_stata` and `StataReader` when loading data from a URL ([GH9231](#)).
- Bug in adding `offsets.Nano` to other offsets raises `TypeError` ([GH9284](#))
- Bug in `DatetimeIndex` iteration, related to ([GH8890](#)), fixed in ([GH9100](#))
- Bugs in `resample` around DST transitions. This required fixing offset classes so they behave correctly on DST transitions. ([GH5172](#), [GH8744](#), [GH8653](#), [GH9173](#), [GH9468](#)).
- Bug in binary operator method (eg `.mul()`) alignment with integer levels ([GH9463](#)).

- Bug in boxplot, scatter and hexbin plot may show an unnecessary warning ([GH8877](#))
- Bug in subplot with `layout` kw may show unnecessary warning ([GH9464](#))
- Bug in using grouper functions that need passed thru arguments (e.g. `axis`), when using wrapped function (e.g. `fillna`), ([GH9221](#))
- `DataFrame` now properly supports simultaneous `copy` and `dtype` arguments in constructor ([GH9099](#))
- Bug in `read_csv` when using `skiprows` on a file with CR line endings with the `c` engine. ([GH9079](#))
- `isnull` now detects `NaT` in `PeriodIndex` ([GH9129](#))
- Bug in `groupby .nth()` with a multiple column `groupby` ([GH8979](#))
- Bug in `DataFrame.where` and `Series.where` coerce numerics to string incorrectly ([GH9280](#))
- Bug in `DataFrame.where` and `Series.where` raise `ValueError` when string list-like is passed. ([GH9280](#))
- Accessing `Series.str` methods on with non-string values now raises `TypeError` instead of producing incorrect results ([GH9184](#))
- Bug in `DatetimeIndex.__contains__` when index has duplicates and is not monotonic increasing ([GH9512](#))
- Fixed division by zero error for `Series.kurt()` when all values are equal ([GH9197](#))
- Fixed issue in the `xlsxwriter` engine where it added a default 'General' format to cells if no other format was applied. This prevented other row or column formatting being applied. ([GH9167](#))
- Fixes issue with `index_col=False` when `usecols` is also specified in `read_csv`. ([GH9082](#))
- Bug where `wide_to_long` would modify the input stubnames list ([GH9204](#))
- Bug in `to_sql` not storing float64 values using double precision. ([GH9009](#))
- `SparseSeries` and `SparsePanel` now accept zero argument constructors (same as their non-sparse counterparts) ([GH9272](#)).
- Regression in merging `Categorical` and object dtypes ([GH9426](#))
- Bug in `read_csv` with buffer overflows with certain malformed input files ([GH9205](#))
- Bug in `groupby MultiIndex` with missing pair ([GH9049](#), [GH9344](#))
- Fixed bug in `Series.groupby` where grouping on `MultiIndex` levels would ignore the sort argument ([GH9444](#))
- Fix bug in `DataFrame.Groupby` where `sort=False` is ignored in the case of `Categorical` columns. ([GH8868](#))
- Fixed bug with reading CSV files from Amazon S3 on python 3 raising a `TypeError` ([GH9452](#))
- Bug in the Google BigQuery reader where the 'jobComplete' key may be present but False in the query results ([GH8728](#))
- Bug in `Series.values_counts` with excluding NaN for categorical type Series with `dropna=True` ([GH9443](#))
- Fixed missing `numeric_only` option for `DataFrame.std/var/sem` ([GH9201](#))
- Support constructing `Panel` or `Panel4D` with scalar data ([GH8285](#))
- `Series` text representation disconnected from `max_rows/max_columns` ([GH7508](#)).

- Series number formatting inconsistent when truncated ([GH8532](#)).

#### Previous Behavior

```
In [2]: pd.options.display.max_rows = 10
In [3]: s = pd.Series([1,1,1,1,1,1,1,1,1,1,0.9999,1,1]*10)
In [4]: s
Out[4]:
0      1
1      1
2      1
...
127    0.9999
128    1.0000
129    1.0000
Length: 130, dtype: float64
```

#### New Behavior

```
0      1.0000
1      1.0000
2      1.0000
3      1.0000
4      1.0000
...
125    1.0000
126    1.0000
127    0.9999
128    1.0000
129    1.0000
dtype: float64
```

- A Spurious SettingWithCopy Warning was generated when setting a new item in a frame in some cases ([GH8730](#))

The following would previously report a SettingWithCopy Warning.

```
In [1]: df1 = DataFrame({'x': Series(['a', 'b', 'c']), 'y': Series(['d', 'e', 'f'])})
In [2]: df2 = df1[['x']]
In [3]: df2['y'] = ['g', 'h', 'i']
```

## 1.20 v0.15.2 (December 12, 2014)

This is a minor release from 0.15.1 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. A small number of API changes were necessary to fix existing bugs. We recommend that all users upgrade to this version.

- *Enhancements*
- *API Changes*
- *Performance Improvements*
- *Bug Fixes*

### 1.20.1 API changes

- Indexing in `MultiIndex` beyond lex-sort depth is now supported, though a lexically sorted index will have a better performance. (GH2646)

```
In [1]: df = pd.DataFrame({'jim':[0, 0, 1, 1],
...:                        'joe':['x', 'x', 'z', 'y'],
...:                        'jolie':np.random.rand(4)}).set_index(['jim', 'joe'])
...:
```

```
In [2]: df
```

Out [2]:

		jolie
jim	joe	
0	x	0.123943
	x	0.119381
1	z	0.738523
	y	0.587304

```
In [3]: df.index.lexsort_depth
```



```
# in prior versions this would raise a KeyError
# will now show a PerformanceWarning
```

```
In [4]: df.loc[(1, 'z')]
```

	jolie
jim joe	
1 z	0.738523

```
# lexically sorting
```

```
In [5]: df2 = df.sort_index()
```

```
In [6]: df2
```

Out [6]:

```
jim joe      jolie
0    x      0.123943
      x      0.119381
1    y      0.587304
      z      0.738523
```

```
In [7]: df2.index.lexsort_depth
```

```
In [8]: df2.loc[(1, 'z')]
```

```
jim      jodie
jim joe
1      z      0.738523
```

- Bug in `unique` of Series with `category` dtype, which returned all categories regardless whether they were “used” or not (see [GH8559](#) for the discussion). Previous behaviour was to return all categories:



```
In [3]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])

In [4]: cat
Out[4]:
[a, b, a]
Categories (3, object): [a < b < c]

In [5]: cat.unique()
Out[5]: array(['a', 'b', 'c'], dtype=object)
```

Now, only the categories that do effectively occur in the array are returned:

```
In [9]: cat = pd.Categorical(['a', 'b', 'a'], categories=['a', 'b', 'c'])

In [10]: cat.unique()
Out[10]:
[a, b]
Categories (2, object): [a, b]
```

- `Series.all` and `Series.any` now support the `level` and `skipna` parameters. `Series.all`, `Series.any`, `Index.all`, and `Index.any` no longer support the `out` and `keepdims` parameters, which existed for compatibility with `ndarray`. Various index types no longer support the `all` and `any` aggregation functions and will now raise `TypeError`. (GH8302).
- Allow equality comparisons of `Series` with a categorical dtype and object dtype; previously these would raise `TypeError` (GH8938)
- Bug in `NDFrame`: conflicting attribute/column names now behave consistently between getting and setting. Previously, when both a column and attribute named `y` existed, `data.y` would return the attribute, while `data.y = z` would update the column (GH8994)

```
In [11]: data = pd.DataFrame({'x': [1, 2, 3]})

In [12]: data.y = 2

In [13]: data['y'] = [2, 4, 6]

In [14]: data
Out[14]:
   x  y
0  1  2
1  2  4
2  3  6

# this assignment was inconsistent
In [15]: data.y = 5
```

Old behavior:

```
In [6]: data.y
Out[6]: 2

In [7]: data['y'].values
Out[7]: array([5, 5, 5])
```

New behavior:

```
In [16]: data.y
Out[16]: 5

In [17]: data['y'].values
Out[17]: array([2, 4, 6])
```

- `Timestamp('now')` is now equivalent to `Timestamp.now()` in that it returns the local time rather than UTC. Also, `Timestamp('today')` is now equivalent to `Timestamp.today()` and both have `tz` as a possible argument. (GH9000)
- Fix negative step support for label-based slices (GH8753)

Old behavior:

```
In [1]: s = pd.Series(np.arange(3), ['a', 'b', 'c'])
Out[1]:
a    0
b    1
c    2
dtype: int64

In [2]: s.loc['c':'a':-1]
Out[2]:
c    2
dtype: int64
```

New behavior:

```
In [18]: s = pd.Series(np.arange(3), ['a', 'b', 'c'])

In [19]: s.loc['c':'a':-1]
Out[19]:
c    2
b    1
a    0
dtype: int64
```

## 1.20.2 Enhancements

Categorical enhancements:

- Added ability to export Categorical data to Stata (GH8633). See [here](#) for limitations of categorical variables exported to Stata data files.
- Added flag `order_categoricals` to `StataReader` and `read_stata` to select whether to order imported categorical data (GH8836). See [here](#) for more information on importing categorical variables from Stata data files.
- Added ability to export Categorical data to to/from HDF5 (GH7621). Queries work the same as if it was an object array. However, the `category` dtyped data is stored in a more efficient manner. See [here](#) for an example and caveats w.r.t. prior versions of pandas.
- Added support for `searchsorted()` on *Categorical* class (GH8420).

Other enhancements:

- Added the ability to specify the SQL type of columns when writing a `DataFrame` to a database (GH8778). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```
from sqlalchemy.types import String
data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

- `Series.all` and `Series.any` now support the `level` and `skipna` parameters (GH8302):

```
In [20]: s = pd.Series([False, True, False], index=[0, 0, 1])

In [21]: s.any(level=0)
Out[21]:
0      True
1     False
dtype: bool
```

- `Panel` now supports the `all` and `any` aggregation functions. (GH8302):

```
In [22]: p = pd.Panel(np.random.rand(2, 5, 4) > 0.1)

In [23]: p.all()
Out[23]:
      0      1      2      3
0  True  True  True  True
1  True  True  False True
2  True  True  True  True
3  True  True  False True
4  True  True  True  True
```

- Added support for `utcfromtimestamp()`, `fromtimestamp()`, and `combine()` on `Timestamp` class (GH5351).
- Added Google Analytics (*pandas.io.ga*) basic documentation (GH8835). See [here](#).
- `Timedelta` arithmetic returns `NotImplemented` in unknown cases, allowing extensions by custom classes (GH8813).
- `Timedelta` now supports arithmetic with `numpy.ndarray` objects of the appropriate dtype (numpy 1.8 or newer only) (GH8884).
- Added `Timedelta.to_timedelta64()` method to the public API (GH8884).
- Added `gbq.generate_bq_schema()` function to the `gbq` module (GH8325).
- `Series` now works with `map` objects the same way as generators (GH8909).
- Added context manager to `HDFStore` for automatic closing (GH8791).
- `to_datetime` gains an `exact` keyword to allow for a format to not require an exact match for a provided format string (if its `False`). `exact` defaults to `True` (meaning that exact matching is still the default) (GH8904)
- Added `axvlines` boolean option to `parallel_coordinates` plot function, determines whether vertical lines will be printed, default is `True`
- Added ability to read table footers to `read_html` (GH8552)
- `to_sql` now infers datatypes of non-NA values for columns that contain NA values and have dtype object (GH8778).

### 1.20.3 Performance

- Reduce memory usage when `skiprows` is an integer in `read_csv` (GH8681)

- Performance boost for `to_datetime` conversions with a passed `format=`, and the `exact=False` (GH8904)

## 1.20.4 Bug Fixes

- Bug in concat of Series with `category` dtype which were coercing to object. (GH8641)
- Bug in Timestamp-Timestamp not returning a Timedelta type and datelike-datelike ops with timezones (GH8865)
- Made consistent a timezone mismatch exception (either tz operated with None or incompatible timezone), will now return `TypeError` rather than `ValueError` (a couple of edge cases only), (GH8865)
- Bug in using a `pd.Grouper(key=...)` with no level/axis or level only (GH8795, GH8866)
- Report a `TypeError` when invalid/no parameters are passed in a groupby (GH8015)
- Bug in packaging pandas with py2app/cx\_Freeze (GH8602, GH8831)
- Bug in groupby signatures that didn't include `*args` or `**kwargs` (GH8733).
- `io.data.Options` now raises `RemoteDataError` when no expiry dates are available from Yahoo and when it receives no data from Yahoo (GH8761), (GH8783).
- Unclear error message in csv parsing when passing dtype and names and the parsed data is a different data type (GH8833)
- Bug in slicing a multi-index with an empty list and at least one boolean indexer (GH8781)
- `io.data.Options` now raises `RemoteDataError` when no expiry dates are available from Yahoo (GH8761).
- `Timedelta` kwargs may now be numpy ints and floats (GH8757).
- Fixed several outstanding bugs for `Timedelta` arithmetic and comparisons (GH8813, GH5963, GH5436).
- `sql_schema` now generates dialect appropriate `CREATE TABLE` statements (GH8697)
- `slice` string method now takes step into account (GH8754)
- Bug in `BlockManager` where setting values with different type would break block integrity (GH8850)
- Bug in `DatetimeIndex` when using time object as key (GH8667)
- Bug in merge where `how='left'` and `sort=False` would not preserve left frame order (GH7331)
- Bug in `MultiIndex.reindex` where reindexing at level would not reorder labels (GH4088)
- Bug in certain operations with dateutil timezones, manifesting with dateutil 2.3 (GH8639)
- Regression in `DatetimeIndex` iteration with a Fixed/Local offset timezone (GH8890)
- Bug in `to_datetime` when parsing a nanoseconds using the `%f` format (GH8989)
- `io.data.Options` now raises `RemoteDataError` when no expiry dates are available from Yahoo and when it receives no data from Yahoo (GH8761), (GH8783).
- Fix: The font size was only set on x axis if vertical or the y axis if horizontal. (GH8765)
- Fixed division by 0 when reading big csv files in python 3 (GH8621)
- Bug in outputting a Multindex with `to_html, index=False` which would add an extra column (GH8452)
- Imported categorical variables from Stata files retain the ordinal information in the underlying data (GH8836).
- Defined `.size` attribute across `NDFrame` objects to provide compat with numpy `>= 1.9.1`; buggy with `np.array_split` (GH8846)

- Skip testing of histogram plots for matplotlib  $\leq 1.2$  (GH8648).
- Bug where `get_data_google` returned object dtypes (GH3995)
- Bug in `DataFrame.stack(..., dropna=False)` when the `DataFrame`'s columns is a `MultiIndex` whose labels do not reference all its levels. (GH8844)
- Bug in that `Option` context applied on `__enter__` (GH8514)
- Bug in `resample` that causes a `ValueError` when resampling across multiple days and the last offset is not calculated from the start of the range (GH8683)
- Bug where `DataFrame.plot(kind='scatter')` fails when checking if an `np.array` is in the `DataFrame` (GH8852)
- Bug in `pd.infer_freq/DataFrame.inferred_freq` that prevented proper sub-daily frequency inference when the index contained DST days (GH8772).
- Bug where index name was still used when plotting a series with `use_index=False` (GH8558).
- Bugs when trying to stack multiple columns, when some (or all) of the level names are numbers (GH8584).
- Bug in `MultiIndex` where `__contains__` returns wrong result if index is not lexically sorted or unique (GH7724)
- BUG CSV: fix problem with trailing whitespace in skipped rows, (GH8679), (GH8661), (GH8983)
- Regression in `Timestamp` does not parse 'Z' zone designator for UTC (GH8771)
- Bug in `StataWriter` the produces writes strings with 244 characters irrespective of actual size (GH8969)
- Fixed `ValueError` raised by `cummin/cummax` when `datetime64` Series contains `NaT`. (GH8965)
- Bug in `Datareader` returns object dtype if there are missing values (GH8980)
- Bug in plotting if `sharex` was enabled and index was a timeseries, would show labels on multiple axes (GH3964).
- Bug where passing a unit to the `TimedeltaIndex` constructor applied the to nano-second conversion twice. (GH9011).
- Bug in plotting of a period-like array (GH9012)

## 1.21 v0.15.1 (November 9, 2014)

This is a minor bug-fix release from 0.15.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- *Enhancements*
- *API Changes*
- *Bug Fixes*

### 1.21.1 API changes

- `s.dt.hour` and other `.dt` accessors will now return `np.nan` for missing values (rather than previously -1), (GH8689)

```
In [1]: s = Series(date_range('20130101', periods=5, freq='D'))

In [2]: s.iloc[2] = np.nan

In [3]: s
Out[3]:
0    2013-01-01
1    2013-01-02
2             NaT
3    2013-01-04
4    2013-01-05
dtype: datetime64[ns]
```

previous behavior:

```
In [6]: s.dt.hour
Out[6]:
0      0
1      0
2     -1
3      0
4      0
dtype: int64
```

current behavior:

```
In [4]: s.dt.hour
Out[4]:
0      0.0
1      0.0
2      NaN
3      0.0
4      0.0
dtype: float64
```

- `groupby` with `as_index=False` will not add erroneous extra columns to result (GH8582):

```
In [5]: np.random.seed(2718281)

In [6]: df = pd.DataFrame(np.random.randint(0, 100, (10, 2)),
...:                      columns=['jim', 'joe'])
...:

In [7]: df.head()
Out[7]:
   jim  joe
0    61   81
1    96   49
2    55   65
3    72   51
4    77   12

In [8]: ts = pd.Series(5 * np.random.randint(0, 3, 10))
```

previous behavior:

```
In [4]: df.groupby(ts, as_index=False).max()
Out[4]:
```

	NaN	jim	joe
0	0	72	83
1	5	77	84
2	10	96	65

current behavior:

```
In [9]: df.groupby(ts, as_index=False).max()
Out[9]:
```

	jim	joe
0	72	83
1	77	84
2	96	65

- groupby will not erroneously exclude columns if the column name conflicts with the grouper name (GH8112):

```
In [10]: df = pd.DataFrame({'jim': range(5), 'joe': range(5, 10)})
In [11]: df
Out[11]:
```

	jim	joe
0	0	5
1	1	6
2	2	7
3	3	8
4	4	9

```
In [12]: gr = df.groupby(df['jim'] < 2)
```

previous behavior (excludes 1st column from output):

```
In [4]: gr.apply(sum)
Out[4]:
```

	joe
jim	
False	24
True	11

current behavior:

```
In [13]: gr.apply(sum)
Out[13]:
```

	jim	joe
jim		
False	9	24
True	1	11

- Support for slicing with monotonic decreasing indexes, even if start or stop is not found in the index (GH7860):

```
In [14]: s = pd.Series(['a', 'b', 'c', 'd'], [4, 3, 2, 1])
In [15]: s
Out[15]:
```

	a
4	a

(continues on next page)

(continued from previous page)

```
3    b
2    c
1    d
dtype: object
```

previous behavior:

```
In [8]: s.loc[3.5:1.5]
KeyError: 3.5
```

current behavior:

```
In [16]: s.loc[3.5:1.5]
Out[16]:
3    b
2    c
dtype: object
```

- `io.data.Options` has been fixed for a change in the format of the Yahoo Options page ([GH8612](#)), ([GH8741](#))

**Note:** As a result of a change in Yahoo's option page layout, when an expiry date is given, `Options` methods now return data for a single expiry date. Previously, methods returned all data for the selected month.

The `month` and `year` parameters have been undeprecated and can be used to get all options data for a given month.

If an expiry date that is not valid is given, data for the next expiry after the given date is returned.

Option data frames are now saved on the instance as `callsYYMMDD` or `putsYYMMDD`. Previously they were saved as `callsSMMYY` and `putsSMMYY`. The next expiry is saved as `calls` and `puts`.

New features:

- The expiry parameter can now be a single date or a list-like object containing dates.
- A new property `expiry_dates` was added, which returns all available expiry dates.

Current behavior:

```
In [17]: from pandas.io.data import Options

In [18]: aapl = Options('aapl', 'yahoo')

In [19]: aapl.get_call_data().iloc[0:5,0:1]
Out[19]:
```

Strike	Expiry	Type	Symbol	Last
80	2014-11-14	call	AAPL141114C00080000	29.05
84	2014-11-14	call	AAPL141114C00084000	24.80
85	2014-11-14	call	AAPL141114C00085000	24.05
86	2014-11-14	call	AAPL141114C00086000	22.76
87	2014-11-14	call	AAPL141114C00087000	21.74

```
In [20]: aapl.expiry_dates
Out[20]:
[datetime.date(2014, 11, 14),
```

(continues on next page)



(continued from previous page)

```

datetime.date(2014, 11, 22),
datetime.date(2014, 11, 28),
datetime.date(2014, 12, 5),
datetime.date(2014, 12, 12),
datetime.date(2014, 12, 20),
datetime.date(2015, 1, 17),
datetime.date(2015, 2, 20),
datetime.date(2015, 4, 17),
datetime.date(2015, 7, 17),
datetime.date(2016, 1, 15),
datetime.date(2017, 1, 20)]

```

```
In [21]: aapl.get_near_stock_price(expiry=aapl.expiry_dates[0:3]).iloc[0:5,0:1]
```

```
Out [21]:
```

	Strike	Expiry	Type	Symbol	Last
109		2014-11-22	call	AAPL141122C00109000	1.48
		2014-11-28	call	AAPL141128C00109000	1.79
110		2014-11-14	call	AAPL141114C00110000	0.55
		2014-11-22	call	AAPL141122C00110000	1.02
		2014-11-28	call	AAPL141128C00110000	1.32

- pandas now also registers the `datetime64` dtype in matplotlib's units registry to plot such values as datetimes. This is activated once pandas is imported. In previous versions, plotting an array of `datetime64` values will have resulted in plotted integer values. To keep the previous behaviour, you can do `del matplotlib.units.registry[np.datetime64]` ([GH8614](#)).

## 1.21.2 Enhancements

- `concat` permits a wider variety of iterables of pandas objects to be passed as the first parameter ([GH8645](#)):

```
In [17]: from collections import deque
```

```
In [18]: df1 = pd.DataFrame([1, 2, 3])
```

```
In [19]: df2 = pd.DataFrame([4, 5, 6])
```

previous behavior:

```
In [7]: pd.concat(deque((df1, df2)))
```

```
TypeError: first argument must be a list-like of pandas objects, you passed an_
↪object of type "deque"
```

current behavior:

```
In [20]: pd.concat(deque((df1, df2)))
```

```
Out [20]:
```

```

0
0 1
1 2
2 3
0 4
1 5
2 6

```

- Represent `MultiIndex` labels with a dtype that utilizes memory based on the level size. In prior versions, the memory usage was a constant 8 bytes per element in each level. In addition, in prior versions, the *reported* memory usage was incorrect as it didn't show the usage for the memory occupied by the underlying data array. (GH8456)

```
In [21]: dfi = DataFrame(1, index=pd.MultiIndex.from_product(['a'], range(1000)),
↳ columns=['A'])
```

previous behavior:

```
# this was underreported in prior versions
In [1]: dfi.memory_usage(index=True)
Out [1]:
Index      8000 # took about 24008 bytes in < 0.15.1
A          8000
dtype: int64
```

current behavior:

```
In [22]: dfi.memory_usage(index=True)
Out [22]:
Index      52080
A          8000
dtype: int64
```

- Added Index properties `is_monotonic_increasing` and `is_monotonic_decreasing` (GH8680).
- Added option to select columns when importing Stata files (GH7935)
- Qualify memory usage in `DataFrame.info()` by adding + if it is a lower bound (GH8578)
- Raise errors in certain aggregation cases where an argument such as `numeric_only` is not handled (GH8592).
- Added support for 3-character ISO and non-standard country codes in `io.wb.download()` (GH8482)
- World Bank data requests now will warn/raise based on an `errors` argument, as well as a list of hard-coded country codes and the World Bank's JSON response. In prior versions, the error messages didn't look at the World Bank's JSON response. Problem-inducing input were simply dropped prior to the request. The issue was that many good countries were cropped in the hard-coded approach. All countries will work now, but some bad countries will raise exceptions because some edge cases break the entire response. (GH8482)
- Added option to `Series.str.split()` to return a `DataFrame` rather than a `Series` (GH8428)
- Added option to `df.info(null_counts=None|True|False)` to override the default display options and force showing of the null-counts (GH8701)

### 1.21.3 Bug Fixes

- Bug in unpickling of a `CustomBusinessDay` object (GH8591)
- Bug in coercing `Categorical` to a records array, e.g. `df.to_records()` (GH8626)
- Bug in `Categorical` not created properly with `Series.to_frame()` (GH8626)
- Bug in coercing in `astype` of a `Categorical` of a passed `pd.Categorical` (this now raises `TypeError` correctly), (GH8626)
- Bug in `cut/qcut` when using `Series` and `retbins=True` (GH8589)
- Bug in writing `Categorical` columns to an SQL database with `to_sql` (GH8624).

- Bug in comparing `Categorical` of datetime raising when being compared to a scalar datetime (GH8687)
- Bug in selecting from a `Categorical` with `.iloc` (GH8623)
- Bug in groupby-transform with a `Categorical` (GH8623)
- Bug in duplicated/drop\_duplicates with a `Categorical` (GH8623)
- Bug in `Categorical` reflected comparison operator raising if the first argument was a numpy array scalar (e.g. `np.int64`) (GH8658)
- Bug in Panel indexing with a list-like (GH8710)
- Compat issue is `DataFrame.dtypes` when `options.mode.use_inf_as_null` is `True` (GH8722)
- Bug in `read_csv`, `dialect` parameter would not take a string (GH8703)
- Bug in slicing a multi-index level with an empty-list (GH8737)
- Bug in numeric index operations of add/sub with `Float/Int Index` with numpy arrays (GH8608)
- Bug in `setitem` with empty indexer and unwanted coercion of dtypes (GH8669)
- Bug in `ix/loc` block splitting on `setitem` (manifests with integer-like dtypes, e.g. `datetime64`) (GH8607)
- Bug when doing label based indexing with integers not found in the index for non-unique but monotonic indexes (GH8680).
- Bug when indexing a `Float64Index` with `np.nan` on numpy 1.7 (GH8980).
- Fix `shape` attribute for `MultiIndex` (GH8609)
- Bug in `GroupBy` where a name conflict between the grouper and columns would break groupby operations (GH7115, GH8112)
- Fixed a bug where plotting a column `y` and specifying a label would mutate the index name of the original `DataFrame` (GH8494)
- Fix regression in plotting of a `DatetimeIndex` directly with `matplotlib` (GH8614).
- Bug in `date_range` where partially-specified dates would incorporate current date (GH6961)
- Bug in Setting by indexer to a scalar value with a mixed-dtype `Panel4d` was failing (GH8702)
- Bug where `DataReader`'s would fail if one of the symbols passed was invalid. Now returns data for valid symbols and `np.nan` for invalid (GH8494)
- Bug in `get_quote_yahoo` that wouldn't allow non-float return values (GH5229).

## 1.22 v0.15.0 (October 18, 2014)

This is a major release from 0.14.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

**Warning:** pandas >= 0.15.0 will no longer support compatibility with NumPy versions < 1.7.0. If you want to use the latest versions of pandas, please upgrade to NumPy >= 1.7.0 (GH7711)

- Highlights include:
  - The `Categorical` type was integrated as a first-class pandas type, see [here](#)
  - New scalar type `Timedelta`, and a new index type `TimedeltaIndex`, see [here](#)

- New datetimelike properties accessor `.dt` for Series, see [Datetimelike Properties](#)
  - New DataFrame default display for `df.info()` to include memory usage, see [Memory Usage](#)
  - `read_csv` will now by default ignore blank lines when parsing, see [here](#)
  - API change in using Indexes in set operations, see [here](#)
  - Enhancements in the handling of timezones, see [here](#)
  - A lot of improvements to the rolling and expanding moment functions, see [here](#)
  - Internal refactoring of the `Index` class to no longer sub-class `ndarray`, see [Internal Refactoring](#)
  - dropping support for PyTables less than version 3.0.0, and numexpr less than version 2.1 (GH7990)
  - Split indexing documentation into [Indexing and Selecting Data](#) and [MultiIndex / Advanced Indexing](#)
  - Split out string methods documentation into [Working with Text Data](#)
- Check the [API Changes](#) and [deprecations](#) before updating
  - [Other Enhancements](#)
  - [Performance Improvements](#)
  - [Bug Fixes](#)

**Warning:** In 0.15.0 `Index` has internally been refactored to no longer sub-class `ndarray` but instead subclass `PandasObject`, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (See the [Internal Refactoring](#))

**Warning:** The refactorings in [Categorical](#) changed the two argument constructor from “codes/labels and levels” to “values and levels (now called ‘categories’)”. This can lead to subtle bugs. If you use [Categorical](#) directly, please audit your code before updating to this pandas version and change it to use the `from_codes()` constructor. See more on [Categorical](#) [here](#)

## 1.22.1 New features

### 1.22.1.1 Categoricals in Series/DataFrame

[Categorical](#) can now be included in *Series* and *DataFrames* and gained new methods to manipulate. Thanks to Jan Schulz for much of this API/implementation. (GH3943, GH5313, GH5314, GH7444, GH7839, GH7848, GH7864, GH7914, GH7768, GH8006, GH3678, GH8075, GH8076, GH8143, GH8453, GH8518).

For full docs, see the [categorical introduction](#) and the [API documentation](#).

```
In [1]: df = DataFrame({"id": [1, 2, 3, 4, 5, 6], "raw_grade": ['a', 'b', 'b', 'a', 'a', 'e']})
In [2]: df["grade"] = df["raw_grade"].astype("category")
In [3]: df["grade"]
Out[3]:
0    a
1    b
```

(continues on next page)

(continued from previous page)

```

2     b
3     a
4     a
5     e
Name: grade, dtype: category
Categories (3, object): [a, b, e]

# Rename the categories
In [4]: df["grade"].cat.categories = ["very good", "good", "very bad"]

# Reorder the categories and simultaneously add the missing categories
In [5]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium",
↳ "good", "very good"])

In [6]: df["grade"]
Out[6]:
0     very good
1         good
2         good
3     very good
4     very good
5     very bad
Name: grade, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]

In [7]: df.sort_values("grade")
////////////////////////////////////
↳
   id raw_grade  grade
5   6         e  very bad
1   2         b    good
2   3         b    good
0   1         a  very good
3   4         a  very good
4   5         a  very good

In [8]: df.groupby("grade").size()
////////////////////////////////////
↳
grade
very bad    1
bad         0
medium      0
good        2
very good   3
dtype: int64

```

- `pandas.core.groupby` and `pandas.core.factor_agg` were removed. As an alternative, construct a dataframe and use `df.groupby(<group>).agg(<func>)`.
- Supplying “codes/labels and levels” to the *Categorical* constructor is not supported anymore. Supplying two arguments to the constructor is now interpreted as “values and levels (now called ‘categories’)”. Please change your code to use the *from\_codes()* constructor.
- The `Categorical.labels` attribute was renamed to `Categorical.codes` and is read only. If you want to manipulate codes, please use one of the *API methods on Categoricals*.
- The `Categorical.levels` attribute is renamed to `Categorical.categories`.

### 1.22.1.2 TimedeltaIndex/Scalar

We introduce a new scalar type `Timedelta`, which is a subclass of `datetime.timedelta`, and behaves in a similar manner, but allows compatibility with `np.timedelta64` types as well as a host of custom representation, parsing, and attributes. This type is very similar to how `Timestamp` works for datetimes. It is a nice-API box for the type. See the *docs*. ([GH3009](#), [GH4533](#), [GH8209](#), [GH8187](#), [GH8190](#), [GH7869](#), [GH7661](#), [GH8345](#), [GH8471](#))

**Warning:** `Timedelta` scalars (and `TimedeltaIndex`) component fields are *not the same* as the component fields on a `datetime.timedelta` object. For example, `.seconds` on a `datetime.timedelta` object returns the total number of seconds combined between hours, minutes and seconds. In contrast, the pandas `Timedelta` breaks out hours, minutes, microseconds and nanoseconds separately.

```
# Timedelta accessor
In [9]: tds = Timedelta('31 days 5 min 3 sec')

In [10]: tds.minutes
Out[10]: 5L

In [11]: tds.seconds
Out[11]: 3L

# datetime.timedelta accessor
# this is 5 minutes * 60 + 3 seconds
In [12]: tds.to_pytimedelta().seconds
Out[12]: 303
```

**Note:** this is no longer true starting from v0.16.0, where full compatibility with `datetime.timedelta` is introduced. See the [0.16.0 whatsnew entry](#)

**Warning:** Prior to 0.15.0 `pd.to_timedelta` would return a `Series` for list-like/`Series` input, and a `np.timedelta64` for scalar input. It will now return a `TimedeltaIndex` for list-like input, `Series` for `Series` input, and `Timedelta` for scalar input.

The arguments to `pd.to_timedelta` are now `(arg, unit='ns', box=True, coerce=False)`, previously were `(arg, box=True, unit='ns')` as these are more logical.

Construct a scalar

```
In [9]: Timedelta('1 days 06:05:01.00003')
Out[9]: Timedelta('1 days 06:05:01.000030')

In [10]: Timedelta('15.5us')
\\Out[10]: Timedelta('0 days 00:00:00.000015
↪')

In [11]: Timedelta('1 hour 15.5us')
\\Out[11]:
↪Timedelta('0 days 01:00:00.000015')

# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [12]: Timedelta('-1us')
\\
↪Timedelta('-1 days +23:59:59.999999')
```

(continues on next page)

(continued from previous page)

```
# a NaT
In [13]: Timedelta('nan')
////////////////////////////////////
↪NaT
```

## Access fields for a Timedelta

```
In [14]: td = Timedelta('1 hour 3m 15.5us')
```

```
In [15]: td.seconds
```

Out[15]: 3780

```
In [16]: td.microseconds
```

```
Out[16]: 16
```

```
In [17]: td.nanoseconds
```

```
Out[17]: 500
```

## Construct a TimedeltaIndex

```
In [18]: TimedeltaIndex(['1 days', '1 days, 00:00:05',
.....:                  np.timedelta64(2, 'D'), timedelta(days=2, seconds=2)])
.....:
```

Out [18] :

```
TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
               '2 days 00:00:02'],
              dtype='timedelta64[ns]', freq=None)
```

## Constructing a TimedeltaIndex with a regular range

```
In [19]: timedelta_range('1 days', periods=5, freq='D')
```

```
Out[19]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
→ 'timedelta64[ns]', freq='D')
```

```
In [20]: timedelta_range(start='1 days', end='2 days', freq='30T')
```

```
TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
                '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                '1 days 10:30:00', '1 days 11:00:00', '1 days 11:30:00',
                '1 days 12:00:00', '1 days 12:30:00', '1 days 13:00:00',
                '1 days 13:30:00', '1 days 14:00:00', '1 days 14:30:00',
                '1 days 15:00:00', '1 days 15:30:00', '1 days 16:00:00',
                '1 days 16:30:00', '1 days 17:00:00', '1 days 17:30:00',
                '1 days 18:00:00', '1 days 18:30:00', '1 days 19:00:00',
                '1 days 19:30:00', '1 days 20:00:00', '1 days 20:30:00',
                '1 days 21:00:00', '1 days 21:30:00', '1 days 22:00:00',
                '1 days 22:30:00', '1 days 23:00:00', '1 days 23:30:00',
                '2 days 00:00:00'],
                dtype='timedelta64[ns]', freq='30T')
```

You can now use a `TimedeltaIndex` as the index of a pandas object

```
In [21]: s = Series(np.arange(5),
.....:             index=timedelta_range('1 days', periods=5, freq='s'))
.....:

In [22]: s
Out[22]:
1 days 00:00:00    0
1 days 00:00:01    1
1 days 00:00:02    2
1 days 00:00:03    3
1 days 00:00:04    4
Freq: S, dtype: int64
```

You can select with partial string selections

```
In [23]: s['1 day 00:00:02']
Out[23]: 2

In [24]: s['1 day':'1 day 00:00:02']
Out[24]:
1 days 00:00:00    0
1 days 00:00:01    1
1 days 00:00:02    2
Freq: S, dtype: int64
```

Finally, the combination of `TimedeltaIndex` with `DatetimeIndex` allow certain combination operations that are NaT preserving:

```
In [25]: tdi = TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [26]: tdi.tolist()
Out[26]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [27]: dti = date_range('20130101', periods=3)

In [28]: dti.tolist()
Out[28]: [Timestamp('2013-01-01 00:00:00', freq='D'),
Timestamp('2013-01-02 00:00:00', freq='D'),
Timestamp('2013-01-03 00:00:00', freq='D')]

In [29]: (dti + tdi).tolist()
Out[29]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [30]: (dti - tdi).tolist()
Out[30]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]
```

- iteration of a Series e.g. `list(Series(...))` of `timedelta64[ns]` would prior to v0.15.0 return `np.timedelta64` for each element. These will now be wrapped in `Timedelta`.

### 1.22.1.3 Memory Usage

Implemented methods to find memory usage of a `DataFrame`. See the [FAQ](#) for more. ([GH6852](#)).



A new display option `display.memory_usage` (see [Options and Settings](#)) sets the default behavior of the `memory_usage` argument in the `df.info()` method. By default `display.memory_usage` is `True`.

```
In [31]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
.....:           'complex128', 'object', 'bool']
.....:

In [32]: n = 5000

In [33]: data = dict([ (t, np.random.randint(100, size=n).astype(t))
.....:                  for t in dtypes])
.....:

In [34]: df = DataFrame(data)

In [35]: df['categorical'] = df['object'].astype('category')

In [36]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
int64          5000 non-null int64
float64        5000 non-null float64
datetime64[ns] 5000 non-null datetime64[ns]
timedelta64[ns] 5000 non-null timedelta64[ns]
complex128     5000 non-null complex128
object         5000 non-null object
bool          5000 non-null bool
categorical    5000 non-null category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↳ object(1), timedelta64[ns](1)
memory usage: 289.1+ KB
```

Additionally `memory_usage()` is an available method for a dataframe object which returns the memory usage of each column.

```
In [37]: df.memory_usage(index=True)
Out[37]:
Index          80
int64         40000
float64       40000
datetime64[ns] 40000
timedelta64[ns] 40000
complex128    80000
object        40000
bool          5000
categorical   10920
dtype: int64
```

#### 1.22.1.4 .dt accessor

`Series` has gained an accessor to succinctly return datetime like properties for the *values* of the `Series`, if its a datetime/period like `Series`. (GH7207) This will return a `Series`, indexed like the existing `Series`. See the [docs](#)

```
# datetime
In [38]: s = Series(date_range('20130101 09:10:12', periods=4))
```

(continues on next page)

(continued from previous page)

```
In [39]: s
Out[39]:
0    2013-01-01 09:10:12
1    2013-01-02 09:10:12
2    2013-01-03 09:10:12
3    2013-01-04 09:10:12
dtype: datetime64[ns]
```

```
In [40]: s.dt.hour
```

```

////////////////////////////////////
↪
0     9
1     9
2     9
3     9
dtype: int64
```

```
In [41]: s.dt.second
```

```

////////////////////////////////////
↪
0    12
1    12
2    12
3    12
dtype: int64
```

```
In [42]: s.dt.day
```

```

////////////////////////////////////
↪
0     1
1     2
2     3
3     4
dtype: int64
```

```
In [43]: s.dt.freq
```

```

////////////////////////////////////
↪ 'D'
```

This enables nice expressions like this:

```
In [44]: s[s.dt.day==2]
Out[44]:
1    2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produce tz aware transformations:

```
In [45]: stz = s.dt.tz_localize('US/Eastern')

In [46]: stz
Out[46]:
0    2013-01-01 09:10:12-05:00
1    2013-01-02 09:10:12-05:00
2    2013-01-03 09:10:12-05:00
3    2013-01-04 09:10:12-05:00
```

(continues on next page)



(continued from previous page)

dtype: timedelta64[ns]

**In [55]:** s.dt.days

```

////////////////////////////////////
↪
0    1
1    1
2    1
3    1
dtype: int64

```

**In [56]:** s.dt.seconds

```

////////////////////////////////////
↪
0    5
1    6
2    7
3    8
dtype: int64

```

**In [57]:** s.dt.components

```

////////////////////////////////////
↪
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0      1      0         0         5              0              0              0
1      1      0         0         6              0              0              0
2      1      0         0         7              0              0              0
3      1      0         0         8              0              0              0

```

### 1.22.1.5 Timezone handling improvements

- `tz_localize(None)` for `tz`-aware `Timestamp` and `DatetimeIndex` now removes timezone holding local time, previously this resulted in `Exception` or `TypeError` ([GH7812](#))

**In [58]:** ts = Timestamp('2014-08-01 09:00', tz='US/Eastern')**In [59]:** ts**Out [59]:** Timestamp('2014-08-01 09:00:00-0400', tz='US/Eastern')**In [60]:** ts.tz\_localize(None)

```

////////////////////////////////////Out [60]: ↪
↪Timestamp('2014-08-01 09:00:00')

```

**In [61]:** didx = DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz='US/Eastern')

**In [62]:** didx**Out [62]:**

```

DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
              '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
              '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
              '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
              '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

```

(continues on next page)

(continued from previous page)

```
In [63]: didx.tz_localize(None)
////////////////////////////////////
↪
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
              '2014-08-01 11:00:00', '2014-08-01 12:00:00',
              '2014-08-01 13:00:00', '2014-08-01 14:00:00',
              '2014-08-01 15:00:00', '2014-08-01 16:00:00',
              '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
              dtype='datetime64[ns]', freq='H')
```

- `tz_localize` now accepts the ambiguous keyword which allows for passing an array of bools indicating whether the date belongs in DST or not, 'NaT' for setting transition times to NaT, 'infer' for inferring DST/non-DST, and 'raise' (default) for an `AmbiguousTimeError` to be raised. See [the docs](#) for more details ([GH7943](#))
- `DataFrame.tz_localize` and `DataFrame.tz_convert` now accepts an optional `level` argument for localizing a specific level of a `MultiIndex` ([GH7846](#))
- `Timestamp.tz_localize` and `Timestamp.tz_convert` now raise `TypeError` in error cases, rather than `Exception` ([GH8025](#))
- a timeseries/index localized to UTC when inserted into a `Series/DataFrame` will preserve the UTC timezone (rather than being a naive `datetime64[ns]`) as object `dtype` ([GH8411](#))
- `Timestamp.__repr__` displays `dateutil.tz.tzoffset` info ([GH7907](#))

### 1.22.1.6 Rolling/Expanding Moments improvements

- `rolling_min()`, `rolling_max()`, `rolling_cov()`, and `rolling_corr()` now return objects with all NaN when `len(arg) < min_periods <= window` rather than raising. (This makes all rolling functions consistent in this behavior). ([GH7766](#))

Prior to 0.15.0

```
In [64]: s = Series([10, 11, 12, 13])
```

```
In [15]: rolling_min(s, window=10, min_periods=5)
ValueError: min_periods (5) must be <= window (4)
```

New behavior

```
In [4]: pd.rolling_min(s, window=10, min_periods=5)
Out[4]:
0    NaN
1    NaN
2    NaN
3    NaN
dtype: float64
```

- `rolling_max()`, `rolling_min()`, `rolling_sum()`, `rolling_mean()`, `rolling_median()`, `rolling_std()`, `rolling_var()`, `rolling_skew()`, `rolling_kurt()`, `rolling_quantile()`, `rolling_cov()`, `rolling_corr()`, `rolling_corr_pairwise()`, `rolling_window()`, and `rolling_apply()` with `center=True` previously would return a result of the same structure as the input `arg` with NaN in the final  $(window-1)/2$  entries.

Now the final  $(window-1)/2$  entries of the result are calculated as if the input `arg` were followed by  $(window-1)/2$  NaN values (or with shrinking windows, in the case of `rolling_apply()`). ([GH7925](#), [GH8269](#))

Prior behavior (note final value is NaN):

```
In [7]: rolling_sum(Series(range(4)), window=3, min_periods=0, center=True)
Out [7]:
0      1
1      3
2      6
3     NaN
dtype: float64
```

New behavior (note final value is 5 = sum([2, 3, NaN])):

```
In [7]: rolling_sum(Series(range(4)), window=3, min_periods=0, center=True)
Out [7]:
0      1
1      3
2      6
3      5
dtype: float64
```

- `rolling_window()` now normalizes the weights properly in rolling mean mode (`mean=True`) so that the calculated weighted means (e.g. ‘triang’, ‘gaussian’) are distributed about the same means as those calculated without weighting (i.e. ‘boxcar’). See [the note on normalization](#) for further details. (GH7618)

```
In [65]: s = Series([10.5, 8.8, 11.4, 9.7, 9.3])
```

Behavior prior to 0.15.0:

```
In [39]: rolling_window(s, window=3, win_type='triang', center=True)
Out [39]:
0      NaN
1    6.583333
2    6.883333
3    6.683333
4      NaN
dtype: float64
```

New behavior

```
In [10]: pd.rolling_window(s, window=3, win_type='triang', center=True)
Out [10]:
0      NaN
1    9.875
2   10.325
3   10.025
4      NaN
dtype: float64
```

- Removed `center` argument from all `expanding_` functions (see [list](#)), as the results produced when `center=True` did not make much sense. (GH7925)
- Added optional `ddof` argument to `expanding_cov()` and `rolling_cov()`. The default value of 1 is backwards-compatible. (GH8279)
- Documented the `ddof` argument to `expanding_var()`, `expanding_std()`, `rolling_var()`, and `rolling_std()`. These functions’ support of a `ddof` argument (with a default value of 1) was previously undocumented. (GH8064)

- `ewma()`, `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now interpret `min_periods` in the same manner that the `rolling_*`() and `expanding_*`() functions do: a given result entry will be NaN if the (expanding, in this case) window does not contain at least `min_periods` values. The previous behavior was to set to NaN the `min_periods` entries starting with the first non- NaN value. (GH7977)

Prior behavior (note values start at index 2, which is `min_periods` after index 0 (the index of the first non-empty value)):

```
In [66]: s = Series([1, None, None, None, 2, 3])
```

```
In [51]: ewma(s, com=3., min_periods=2)
```

```
Out [51]:
0      NaN
1      NaN
2    1.000000
3    1.000000
4    1.571429
5    2.189189
dtype: float64
```

New behavior (note values start at index 4, the location of the 2nd (since `min_periods=2`) non-empty value):

```
In [2]: pd.ewma(s, com=3., min_periods=2)
```

```
Out [2]:
0      NaN
1      NaN
2      NaN
3      NaN
4    1.759644
5    2.383784
dtype: float64
```

- `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now have an optional `adjust` argument, just like `ewma()` does, affecting how the weights are calculated. The default value of `adjust` is `True`, which is backwards-compatible. See *Exponentially weighted moment functions* for details. (GH7911)
- `ewma()`, `ewmstd()`, `ewmvol()`, `ewmvar()`, `ewmcov()`, and `ewmcorr()` now have an optional `ignore_na` argument. When `ignore_na=False` (the default), missing values are taken into account in the weights calculation. When `ignore_na=True` (which reproduces the pre-0.15.0 behavior), missing values are ignored in the weights calculation. (GH7543)

```
In [7]: pd.ewma(Series([None, 1., 8.]), com=2.)
```

```
Out [7]:
0      NaN
1     1.0
2     5.2
dtype: float64
```

```
In [8]: pd.ewma(Series([1., None, 8.]), com=2., ignore_na=True) # pre-0.15.0
↪behavior
```

```
Out [8]:
0     1.0
1     1.0
2     5.2
dtype: float64
```

```
In [9]: pd.ewma(Series([1., None, 8.]), com=2., ignore_na=False) # new default
```

(continues on next page)

(continued from previous page)

```
Out [9]:
0      1.000000
1      1.000000
2      5.846154
dtype: float64
```

**Warning:** By default (`ignore_na=False`) the `ewm*()` functions' weights calculation in the presence of missing values is different than in pre-0.15.0 versions. To reproduce the pre-0.15.0 calculation of weights in the presence of missing values one must specify explicitly `ignore_na=True`.

- Bug in `expanding_cov()`, `expanding_corr()`, `rolling_cov()`, `rolling_cor()`, `ewmcov()`, and `ewmcorr()` returning results with columns sorted by name and producing an error for non-unique columns; now handles non-unique columns and returns columns in original order (except for the case of two DataFrames with `pairwise=False`, where behavior is unchanged) (GH7542)
- Bug in `rolling_count()` and `expanding_*()` functions unnecessarily producing error message for zero-length data (GH8056)
- Bug in `rolling_apply()` and `expanding_apply()` interpreting `min_periods=0` as `min_periods=1` (GH8080)
- Bug in `expanding_std()` and `expanding_var()` for a single value producing a confusing error message (GH7900)
- Bug in `rolling_std()` and `rolling_var()` for a single value producing 0 rather than NaN (GH7900)
- Bug in `ewmstd()`, `ewmvol()`, `ewmvar()`, and `ewmcov()` calculation of de-biasing factors when `bias=False` (the default). Previously an incorrect constant factor was used, based on `adjust=True`, `ignore_na=True`, and an infinite number of observations. Now a different factor is used for each entry, based on the actual weights (analogous to the usual  $N/(N-1)$  factor). In particular, for a single point a value of NaN is returned when `bias=False`, whereas previously a value of (approximately) 0 was returned.

For example, consider the following pre-0.15.0 results for `ewmvar(..., bias=False)`, and the corresponding debiasing factors:

```
In [67]: s = Series([1., 2., 0., 4.])
```

```
In [89]: ewmvar(s, com=2., bias=False)
```

```
Out [89]:
0      -2.775558e-16
1       3.000000e-01
2       9.556787e-01
3       3.585799e+00
dtype: float64
```

```
In [90]: ewmvar(s, com=2., bias=False) / ewmvar(s, com=2., bias=True)
```

```
Out [90]:
0       1.25
1       1.25
2       1.25
3       1.25
dtype: float64
```

Note that entry 0 is approximately 0, and the debiasing factors are a constant 1.25. By comparison, the following 0.15.0 results have a NaN for entry 0, and the debiasing factors are decreasing (towards 1.25):



```

In [14]: pd.ewmvar(s, com=2., bias=False)
Out[14]:
0      NaN
1    0.500000
2    1.210526
3    4.089069
dtype: float64

In [15]: pd.ewmvar(s, com=2., bias=False) / pd.ewmvar(s, com=2., bias=True)
Out[15]:
0      NaN
1    2.083333
2    1.583333
3    1.425439
dtype: float64

```

See *Exponentially weighted moment functions* for details. (GH7912)

### 1.22.1.7 Improvements in the sql io module

- Added support for a `chunksize` parameter to `to_sql` function. This allows `DataFrame` to be written in chunks and avoid packet-size overflow errors (GH8062).
- Added support for a `chunksize` parameter to `read_sql` function. Specifying this argument will return an iterator through chunks of the query result (GH2908).
- Added support for writing `datetime.date` and `datetime.time` object columns with `to_sql` (GH6932).
- Added support for specifying a `schema` to read from/write to with `read_sql_table` and `to_sql` (GH7441, GH7952). For example:

```

df.to_sql('table', engine, schema='other_schema')
pd.read_sql_table('table', engine, schema='other_schema')

```

- Added support for writing `NaN` values with `to_sql` (GH2754).
- Added support for writing `datetime64` columns with `to_sql` for all database flavors (GH7103).

## 1.22.2 Backwards incompatible API changes

### 1.22.2.1 Breaking changes

API changes related to `Categorical` (see [here](#) for more details):

- The `Categorical` constructor with two arguments changed from “codes/labels and levels” to “values and levels (now called ‘categories’)”. This can lead to subtle bugs. If you use `Categorical` directly, please audit your code by changing it to use the `from_codes()` constructor.

An old function call like (prior to 0.15.0):

```
pd.Categorical([0,1,0,2,1], levels=['a', 'b', 'c'])
```

will have to adapted to the following to keep the same behaviour:

```
In [2]: pd.Categorical.from_codes([0,1,0,2,1], categories=['a', 'b', 'c'])
Out[2]:
[a, b, a, c, b]
Categories (3, object): [a, b, c]
```

API changes related to the introduction of the Timedelta scalar (see [above](#) for more details):

- Prior to 0.15.0 `to_timedelta()` would return a `Series` for list-like/`Series` input, and a `np.timedelta64` for scalar input. It will now return a `TimedeltaIndex` for list-like input, `Series` for `Series` input, and `Timedelta` for scalar input.

For API changes related to the rolling and expanding functions, see detailed overview [above](#).

Other notable API changes:

- Consistency when indexing with `.loc` and a list-like indexer when no values are found.

```
In [68]: df = DataFrame(['a'], ['b'], index=[1,2])

In [69]: df
Out[69]:
0
1  a
2  b
```

In prior versions there was a difference in these two constructs:

- `df.loc[[3]]` would return a frame reindexed by 3 (with all `np.nan` values)
- `df.loc[[3],:]` would raise `KeyError`.

Both will now raise a `KeyError`. The rule is that *at least 1* indexer must be found when using a list-like and `.loc` ([GH7999](#))

Furthermore in prior versions these were also different:

- `df.loc[[1,3]]` would return a frame reindexed by [1,3]
- `df.loc[[1,3],:]` would raise `KeyError`.

Both will now return a frame reindex by [1,3]. E.g.

```
In [3]: df.loc[[1,3]]
Out[3]:
0
1  a
3 NaN

In [4]: df.loc[[1,3],:]
Out[4]:
0
1  a
3 NaN
```

This can also be seen in multi-axis indexing with a `Panel`.

```
In [70]: p = Panel(np.arange(2*3*4).reshape(2,3,4),
.....:             items=['ItemA', 'ItemB'],
.....:             major_axis=[1,2,3],
.....:             minor_axis=['A', 'B', 'C', 'D'])
.....:
```

(continues on next page)

(continued from previous page)

```
In [71]: p
Out[71]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemB
Major_axis axis: 1 to 3
Minor_axis axis: A to D
```

The following would raise `KeyError` prior to 0.15.0:

```
In [5]:
Out[5]:
```

	ItemA	ItemD
1	3	NaN
2	7	NaN
3	11	NaN

Furthermore, `.loc` will raise `KeyError` if no values are found in a multi-index with a list-like indexer:

[illegible]

- Assigning values to `None` now considers the dtype when choosing an ‘empty’ value ([GH7941](#)).

Previously, assigning to `None` in numeric containers changed the dtype to object (or errored, depending on the call). It now uses `NaN`:

```
In [75]: s = Series([1, 2, 3])

In [76]: s.loc[0] = None

In [77]: s
Out[77]:
0      NaN
1      2.0
2      3.0
dtype: float64
```

NaT is now used similarly for datetime containers.

For object containers, we now preserve `None` values (previously these were converted to NaN values).

```
In [78]: s = Series(["a", "b", "c"])

In [79]: s.loc[0] = None

In [80]: s
Out[80]:
0      None
1         b
2         c
dtype: object
```

To insert a NaN, you must explicitly use `np.nan`. See the [docs](#).

- In prior versions, updating a pandas object inplace would not reflect in other python references to this object. (GH8511, GH5104)

```
In [81]: s = Series([1, 2, 3])

In [82]: s2 = s

In [83]: s += 1.5
```

Behavior prior to v0.15.0

```
# the original object
In [5]: s
Out[5]:
0      2.5
1      3.5
2      4.5
dtype: float64

# a reference to the original object
In [7]: s2
Out[7]:
0      1
1      2
2      3
dtype: int64
```

This is now the correct behavior

```
# the original object
In [84]: s
Out[84]:
0      2.5
1      3.5
2      4.5
dtype: float64

# a reference to the original object
In [85]: s2
Out[85]:
0      2.5
```

(continues on next page)

(continued from previous page)

```
1    3.5
2    4.5
dtype: float64
```

- Made both the C-based and Python engines for `read_csv` and `read_table` ignore empty lines in input as well as whitespace-filled lines, as long as `sep` is not whitespace. This is an API change that can be controlled by the keyword parameter `skip_blank_lines`. See [the docs](#) (GH4466)
- A timeseries/index localized to UTC when inserted into a Series/DataFrame will preserve the UTC timezone and inserted as `object` dtype rather than being converted to a naive `datetime64[ns]` (GH8411).
- Bug in passing a `DatetimeIndex` with a timezone that was not being retained in DataFrame construction from a dict (GH7822)

In prior versions this would drop the timezone, now it retains the timezone, but gives a column of `object` dtype:

```
In [86]: i = date_range('1/1/2011', periods=3, freq='10s', tz = 'US/Eastern')

In [87]: i
Out[87]:
DatetimeIndex(['2011-01-01 00:00:00-05:00', '2011-01-01 00:00:10-05:00',
              '2011-01-01 00:00:20-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='10S')

In [88]: df = DataFrame( {'a' : i } )

In [89]: df
Out[89]:
           a
0 2011-01-01 00:00:00-05:00
1 2011-01-01 00:00:10-05:00
2 2011-01-01 00:00:20-05:00

In [90]: df.dtypes
//////////
↪
a    datetime64[ns, US/Eastern]
dtype: object
```

Previously this would have yielded a column of `datetime64` dtype, but without timezone info.

The behaviour of assigning a column to an existing dataframe as `df['a'] = i` remains unchanged (this already returned an `object` column with a timezone).

- When passing multiple levels to `stack()`, it will now raise a `ValueError` when the levels aren't all level names or all level numbers (GH7660). See [Reshaping by stacking and unstacking](#).
- Raise a `ValueError` in `df.to_hdf` with 'fixed' format, if `df` has non-unique columns as the resulting file will be broken (GH7761)
- `SettingWithCopy` raise/warnings (according to the option `mode.chained_assignment`) will now be issued when setting a value on a sliced mixed-dtype DataFrame using chained-assignment. (GH7845, GH7950)

```
In [1]: df = DataFrame(np.arange(0,9), columns=['count'])

In [2]: df['group'] = 'b'
```

(continues on next page)

(continued from previous page)

```
In [3]: df.iloc[0:5]['group'] = 'a'
/usr/local/bin/ipython:1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/
↪indexing.html#indexing-view-versus-copy
```

- `merge`, `DataFrame.merge`, and `ordered_merge` now return the same type as the left argument (GH7737).
- Previously an enlargement with a mixed-dtype frame would act unlike `.append` which will preserve dtypes (related GH2578, GH8176):

```
In [91]: df = DataFrame([[True, 1], [False, 2]],
.....:                  columns=["female", "fitness"])
.....:

In [92]: df
Out[92]:
   female  fitness
0    True         1
1   False         2

In [93]: df.dtypes
Out[93]:
female      bool
fitness    int64
dtype: object

# dtypes are now preserved
In [94]: df.loc[2] = df.loc[1]

In [95]: df
Out[95]:
   female  fitness
0    True         1
1   False         2
2   False         2

In [96]: df.dtypes
Out[96]:
female      bool
fitness    int64
dtype: object
```

- `Series.to_csv()` now returns a string when `path=None`, matching the behaviour of `DataFrame.to_csv()` (GH8215).
- `read_hdf` now raises `IOError` when a file that doesn't exist is passed in. Previously, a new, empty file was created, and a `KeyError` raised (GH7715).
- `DataFrame.info()` now ends its output with a newline character (GH8114)
- Concatenating no objects will now raise a `ValueError` rather than a bare `Exception`.
- Merge errors will now be sub-classes of `ValueError` rather than raw `Exception` (GH8501)

- `DataFrame.plot` and `Series.plot` keywords are now have consistent orders (GH8037)

### 1.22.2.2 Internal Refactoring

In 0.15.0 `Index` has internally been refactored to no longer sub-class `ndarray` but instead subclass `PandasObject`, similarly to the rest of the pandas objects. This change allows very easy sub-classing and creation of new index types. This should be a transparent change with only very limited API implications (GH5080, GH7439, GH7796, GH8024, GH8367, GH7997, GH8522):

- you may need to unpickle pandas version < 0.15.0 pickles using `pd.read_pickle` rather than `pickle.load`. See [pickle docs](#)
- when plotting with a `PeriodIndex`, the matplotlib internal axes will now be arrays of `Period` rather than a `PeriodIndex` (this is similar to how a `DatetimeIndex` passes arrays of datetimes now)
- `MultiIndexes` will now raise similarly to other pandas objects w.r.t. truth testing, see [here](#) (GH7897).
- When plotting a `DatetimeIndex` directly with matplotlib's `plot` function, the axis labels will no longer be formatted as dates but as integers (the internal representation of a `datetime64`). **UPDATE** This is fixed in 0.15.1, see [here](#).

### 1.22.2.3 Deprecations

- The attributes `Categorical.labels` and `levels` attributes are deprecated and renamed to `codes` and `categories`.
- The `outtype` argument to `pd.DataFrame.to_dict` has been deprecated in favor of `orient`. (GH7840)
- The `convert_dummies` method has been deprecated in favor of `get_dummies` (GH8140)
- The `infer_dst` argument in `tz_localize` will be deprecated in favor of `ambiguous` to allow for more flexibility in dealing with DST transitions. Replace `infer_dst=True` with `ambiguous='infer'` for the same behavior (GH7943). See [the docs](#) for more details.
- The top-level `pd.value_range` has been deprecated and can be replaced by `.describe()` (GH8481)
- The `Index` set operations `+` and `-` were deprecated in order to provide these for numeric type operations on certain index types. `+` can be replaced by `.union()` or `|`, and `-` by `.difference()`. Further the method name `Index.diff()` is deprecated and can be replaced by `Index.difference()` (GH8226)

```
# +
Index(['a', 'b', 'c']) + Index(['b', 'c', 'd'])

# should be replaced by
Index(['a', 'b', 'c']).union(Index(['b', 'c', 'd']))
```

```
# -
Index(['a', 'b', 'c']) - Index(['b', 'c', 'd'])

# should be replaced by
Index(['a', 'b', 'c']).difference(Index(['b', 'c', 'd']))
```

- The `infer_types` argument to `read_html()` now has no effect and is deprecated (GH7762, GH7032).

### 1.22.2.4 Removal of prior version deprecations/changes

- Remove `DataFrame.delevel` method in favor of `DataFrame.reset_index`

- Added support for bool, uint8, uint16 and uint32 datatypes in `to_stata` ([GH7097](#), [GH7365](#))
- Added conversion option when importing Stata files ([GH8527](#))
- `DataFrame.to_stata` and `StataWriter` check string length for compatibility with limitations imposed in dta files where fixed-width strings must contain 244 or fewer characters. Attempting to write Stata dta files with strings longer than 244 characters raises a `ValueError`. ([GH7858](#))
- `read_stata` and `StataReader` can import missing data information into a `DataFrame` by setting the argument `convert_missing` to `True`. When using this options, missing values are returned as `StataMissingValue` objects and columns containing missing values have object data type. ([GH8045](#))

- Added layout keyword to `DataFrame.plot`. You can pass a tuple of (rows, columns), one of which can be -1 to automatically infer ([GH6667](#), [GH8071](#)).
- Allow to pass multiple axes to `DataFrame.plot`, `hist` and `boxplot` ([GH5353](#), [GH6970](#), [GH7069](#))
- Added support for `c`, `colormap` and `colorbar` arguments for `DataFrame.plot` with `kind='scatter'` ([GH7780](#))
- Histogram from `DataFrame.plot` with `kind='hist'` ([GH7809](#)), See [the docs](#).
- Boxplot from `DataFrame.plot` with `kind='box'` ([GH7998](#)), See [the docs](#).

- `read_csv` now has a keyword parameter `float_precision` which specifies which floating-point converter the C engine should use during parsing, see [here](#) (GH8002, GH8044)
- Added `searchsorted` method to `Series` objects (GH7447)
- `describe()` on mixed-types `DataFrames` is more flexible. Type-based column filtering is now possible via the `include/exclude` arguments. See the [docs](#) (GH8164).

```

      catA catB      numC
count    24   24  24.000000
unique     2    4         NaN
top      foo   a         NaN
freq     16    6         NaN

```

## Chapter 1. What's New



(continued from previous page)

mean	NaN	NaN	11.500000
std	NaN	NaN	7.071068
min	NaN	NaN	0.000000
25%	NaN	NaN	5.750000
50%	NaN	NaN	11.500000
75%	NaN	NaN	17.250000
max	NaN	NaN	23.000000

Requesting all columns is possible with the shorthand 'all'

```
In [100]: df.describe(include='all')
Out[100]:
```

	catA	catB	numC	numD
count	24	24	24.000000	24.000000
unique	2	4	NaN	NaN
top	foo	a	NaN	NaN
freq	16	6	NaN	NaN
mean	NaN	NaN	11.500000	12.000000
std	NaN	NaN	7.071068	7.071068
min	NaN	NaN	0.000000	0.500000
25%	NaN	NaN	5.750000	6.250000
50%	NaN	NaN	11.500000	12.000000
75%	NaN	NaN	17.250000	17.750000
max	NaN	NaN	23.000000	23.500000

Without those arguments, describe will behave as before, including only numerical columns or, if none are, only categorical columns. See also the [docs](#)

- Added split as an option to the orient argument in `pd.DataFrame.to_dict`. ([GH7840](#))
- The `get_dummies` method can now be used on DataFrames. By default only catagorical columns are encoded as 0's and 1's, while other columns are left untouched.

```
In [101]: df = DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
.....:                  'C': [1, 2, 3]})
.....:

In [102]: pd.get_dummies(df)
Out[102]:
```

	C	A_a	A_b	B_b	B_c
0	1	1	0	0	1
1	2	0	1	0	1
2	3	1	0	1	0

- `PeriodIndex` supports resolution as the same as `DatetimeIndex` ([GH7708](#))
- `pandas.tseries.holiday` has added support for additional holidays and ways to observe holidays ([GH7070](#))
- `pandas.tseries.holiday.Holiday` now supports a list of offsets in Python3 ([GH7070](#))
- `pandas.tseries.holiday.Holiday` now supports a `days_of_week` parameter ([GH7070](#))
- `GroupBy.nth()` now supports selecting multiple nth values ([GH7910](#))

```
In [103]: business_dates = date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [104]: df = DataFrame(1, index=business_dates, columns=['a', 'b'])
```

(continues on next page)

(continued from previous page)

```
# get the first, 4th, and last date index for each month
In [105]: df.groupby((df.index.year, df.index.month)).nth([0, 3, -1])
Out[105]:
```

	a	b
2014 4	1	1
4	1	1
4	1	1
5	1	1
5	1	1
5	1	1
6	1	1
6	1	1
6	1	1

- Period and PeriodIndex supports addition/subtraction with timedelta-like (GH7966)

If Period freq is D, H, T, S, L, U, N, Timedelta-like can be added if the result can have same freq. Otherwise, only the same offsets can be added.

```
In [106]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')
In [107]: idx
Out[107]:
PeriodIndex(['2014-07-01 09:00', '2014-07-01 10:00', '2014-07-01 11:00',
            '2014-07-01 12:00', '2014-07-01 13:00'],
            dtype='period[H]', freq='H')

In [108]: idx + pd.offsets.Hour(2)
↪
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
            '2014-07-01 14:00', '2014-07-01 15:00'],
            dtype='period[H]', freq='H')

In [109]: idx + Timedelta('120m')
↪
PeriodIndex(['2014-07-01 11:00', '2014-07-01 12:00', '2014-07-01 13:00',
            '2014-07-01 14:00', '2014-07-01 15:00'],
            dtype='period[H]', freq='H')

In [110]: idx = pd.period_range('2014-07', periods=5, freq='M')
In [111]: idx
Out[111]: PeriodIndex(['2014-07', '2014-08', '2014-09', '2014-10', '2014-11'],
↪ dtype='period[M]', freq='M')

In [112]: idx + pd.offsets.MonthEnd(3)
↪
PeriodIndex(['2014-10', '2014-11', '2014-12', '2015-01', '2015-02'], dtype=
↪ 'period[M]', freq='M')
```

- Added experimental compatibility with openpyxl for versions >= 2.0. The DataFrame.to\_excel method engine keyword now recognizes openpyxl1 and openpyxl2 which will explicitly require openpyxl v1 and v2 respectively, failing if the requested version is not available. The openpyxl engine is now a meta-engine that automatically uses whichever version of openpyxl is installed. (GH7177)

- `DataFrame.fillna` can now accept a `DataFrame` as a fill value ([GH8377](#))
- Passing multiple levels to `stack()` will now work when multiple level numbers are passed ([GH7660](#)). See *Reshaping by stacking and unstacking*.
- `set_names()`, `set_labels()`, and `set_levels()` methods now take an optional `level` keyword argument to all modification of specific level(s) of a `MultiIndex`. Additionally `set_names()` now accepts a scalar string value when operating on an `Index` or on a specific level of a `MultiIndex` ([GH7792](#))

```
In [113]: idx = MultiIndex.from_product(['a'], range(3), list("pqr")), names=[
↳ 'foo', 'bar', 'baz'])

In [114]: idx.set_names('qux', level=0)
Out[114]:
MultiIndex(levels=[['a'], [0, 1, 2], ['p', 'q', 'r']],
            labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0,
↳ 1, 2, 0, 1, 2, 0, 1, 2]],
            names=['qux', 'bar', 'baz'])

In [115]: idx.set_names(['qux', 'corge'], level=[0,1])
↳
MultiIndex(levels=[['a'], [0, 1, 2], ['p', 'q', 'r']],
            labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0,
↳ 1, 2, 0, 1, 2, 0, 1, 2]],
            names=['qux', 'corge', 'baz'])

In [116]: idx.set_levels(['a', 'b', 'c'], level='bar')
↳
MultiIndex(levels=[['a'], ['a', 'b', 'c'], ['p', 'q', 'r']],
            labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0,
↳ 1, 2, 0, 1, 2, 0, 1, 2]],
            names=['foo', 'bar', 'baz'])

In [117]: idx.set_levels(['a', 'b', 'c'], [1,2,3], level=[1,2])
↳
MultiIndex(levels=[['a'], ['a', 'b', 'c'], [1, 2, 3]],
            labels=[[0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 1, 1, 2, 2, 2], [0,
↳ 1, 2, 0, 1, 2, 0, 1, 2]],
            names=['foo', 'bar', 'baz'])
```

- `Index.isin` now supports a `level` argument to specify which index level to use for membership tests ([GH7892](#), [GH7890](#))

```
In [1]: idx = MultiIndex.from_product([[0, 1], ['a', 'b', 'c']])

In [2]: idx.values
Out[2]: array([(0, 'a'), (0, 'b'), (0, 'c'), (1, 'a'), (1, 'b'), (1, 'c')],
↳ dtype=object)

In [3]: idx.isin(['a', 'c', 'e'], level=1)
Out[3]: array([ True, False,  True,  True, False,  True], dtype=bool)
```

- `Index` now supports duplicated and `drop_duplicates`. ([GH4060](#))

```
In [118]: idx = Index([1, 2, 3, 4, 1, 2])
```

(continues on next page)

(continued from previous page)

```

In [119]: idx
Out[119]: Int64Index([1, 2, 3, 4, 1, 2], dtype='int64')

In [120]: idx.duplicated()
Out[120]: array([False,  1
↪False, False, False,  True,  True], dtype=bool)

In [121]: idx.drop_duplicates()
Out[121]: Int64Index([1, 2, 3, 4], dtype='int64')

```

- add `copy=True` argument to `pd.concat` to enable pass thru of complete blocks (GH8252)
- Added support for numpy 1.8+ data types (`bool_`, `int_`, `float_`, `string_`) for conversion to R dataframe (GH8400)

## 1.22.4 Performance

- Performance improvements in `DatetimeIndex.__iter__` to allow faster iteration (GH7683)
- Performance improvements in `Period` creation (and `PeriodIndex.setitem`) (GH5155)
- Improvements in `Series.transform` for significant performance gains (revised) (GH6496)
- Performance improvements in `StataReader` when reading large files (GH8040, GH8073)
- Performance improvements in `StataWriter` when writing large files (GH8079)
- Performance and memory usage improvements in multi-key `groupby` (GH8128)
- Performance improvements in `groupby.agg` and `groupby.apply` where builtins `max/min` were not mapped to numpy/cythonized versions (GH7722)
- Performance improvement in writing to sql (`to_sql`) of up to 50% (GH8208).
- Performance benchmarking of `groupby` for large value of `ngroups` (GH6787)
- Performance improvement in `CustomBusinessDay`, `CustomBusinessMonth` (GH8236)
- Performance improvement for `MultiIndex.values` for multi-level indexes containing datetimes (GH8543)

## 1.22.5 Bug Fixes

- Bug in `pivot_table`, when using margins and a dict `aggfunc` (GH8349)
- Bug in `read_csv` where `squeeze=True` would return a view (GH8217)
- Bug in checking of table name in `read_sql` in certain cases (GH7826).
- Bug in `DataFrame.groupby` where `Grouper` does not recognize level when frequency is specified (GH7885)
- Bug in multiindexes dtypes getting mixed up when `DataFrame` is saved to SQL table (GH8021)
- Bug in `Series` 0-division with a float and integer operand dtypes (GH7785)
- Bug in `Series.astype("unicode")` not calling `unicode` on the values correctly (GH7758)
- Bug in `DataFrame.as_matrix()` with mixed `datetime64[ns]` and `timedelta64[ns]` dtypes (GH7778)

- Bug in `HDFStore.select_column()` not preserving UTC timezone info when selecting a `DatetimeIndex` (GH7777)
- Bug in `to_datetime` when `format='%Y%m%d'` and `coerce=True` are specified, where previously an object array was returned (rather than a coerced time-series with `NaT`), (GH7930)
- Bug in `DatetimeIndex` and `PeriodIndex` in-place addition and subtraction cause different result from normal one (GH6527)
- Bug in adding and subtracting `PeriodIndex` with `PeriodIndex` raise `TypeError` (GH7741)
- Bug in `combine_first` with `PeriodIndex` data raises `TypeError` (GH3367)
- Bug in multi-index slicing with missing indexers (GH7866)
- Bug in multi-index slicing with various edge cases (GH8132)
- Regression in multi-index indexing with a non-scalar type object (GH7914)
- Bug in Timestamp comparisons with `==` and `int64` dtype (GH8058)
- Bug in pickles contains `DateOffset` may raise `AttributeError` when `normalize` attribute is referred internally (GH7748)
- Bug in Panel when using `major_xs` and `copy=False` is passed (deprecation warning fails because of missing warnings) (GH8152).
- Bug in pickle deserialization that failed for pre-0.14.1 containers with dup items trying to avoid ambiguity when matching block and manager items, when there's only one block there's no ambiguity (GH7794)
- Bug in putting a `PeriodIndex` into a `Series` would convert to `int64` dtype, rather than object of `Periods` (GH7932)
- Bug in `HDFStore` iteration when passing a `where` (GH8014)
- Bug in `DataFrameGroupby.transform` when transforming with a passed non-sorted key (GH8046, GH8430)
- Bug in repeated timeseries line and area plot may result in `ValueError` or incorrect kind (GH7733)
- Bug in inference in a `MultiIndex` with `datetime.date` inputs (GH7888)
- Bug in `get` where an `IndexError` would not cause the default value to be returned (GH7725)
- Bug in `offsets.apply`, `rollforward` and `rollback` may reset nanosecond (GH7697)
- Bug in `offsets.apply`, `rollforward` and `rollback` may raise `AttributeError` if `Timestamp` has `dateutil.tzinfo` (GH7697)
- Bug in sorting a multi-index frame with a `Float64Index` (GH8017)
- Bug in inconsistent panel setitem with a rhs of a `DataFrame` for alignment (GH7763)
- Bug in `is_superperiod` and `is_subperiod` cannot handle higher frequencies than `S` (GH7760, GH7772, GH7803)
- Bug in 32-bit platforms with `Series.shift` (GH8129)
- Bug in `PeriodIndex.unique` returns `int64 np.ndarray` (GH7540)
- Bug in `groupby.apply` with a non-affecting mutation in the function (GH8467)
- Bug in `DataFrame.reset_index` which has `MultiIndex` contains `PeriodIndex` or `DatetimeIndex` with `tz` raises `ValueError` (GH7746, GH7793)
- Bug in `DataFrame.plot` with `subplots=True` may draw unnecessary minor xticks and yticks (GH7801)

- Bug in `StataReader` which did not read variable labels in 117 files due to difference between Stata documentation and implementation ([GH7816](#))
- Bug in `StataReader` where strings were always converted to 244 characters-fixed width irrespective of underlying string size ([GH7858](#))
- Bug in `DataFrame.plot` and `Series.plot` may ignore `rot` and `fontsize` keywords ([GH7844](#))
- Bug in `DatetimeIndex.value_counts` doesn't preserve `tz` ([GH7735](#))
- Bug in `PeriodIndex.value_counts` results in `Int64Index` ([GH7735](#))
- Bug in `DataFrame.join` when doing left join on index and there are multiple matches ([GH5391](#))
- Bug in `GroupBy.transform()` where int groups with a transform that didn't preserve the index were incorrectly truncated ([GH7972](#)).
- Bug in `groupby` where callable objects without name attributes would take the wrong path, and produce a `DataFrame` instead of a `Series` ([GH7929](#))
- Bug in `groupby` error message when a `DataFrame` grouping column is duplicated ([GH7511](#))
- Bug in `read_html` where the `infer_types` argument forced coercion of date-likes incorrectly ([GH7762](#), [GH7032](#)).
- Bug in `Series.str.cat` with an index which was filtered as to not include the first item ([GH7857](#))
- Bug in `Timestamp` cannot parse nanosecond from string ([GH7878](#))
- Bug in `Timestamp` with string offset and `tz` results incorrect ([GH7833](#))
- Bug in `tslib.tz_convert` and `tslib.tz_convert_single` may return different results ([GH7798](#))
- Bug in `DatetimeIndex.intersection` of non-overlapping timestamps with `tz` raises `IndexError` ([GH7880](#))
- Bug in alignment with `TimeOps` and non-unique indexes ([GH8363](#))
- Bug in `GroupBy.filter()` where fast path vs. slow path made the filter return a non scalar value that appeared valid but wasn't ([GH7870](#)).
- Bug in `date_range()/DatetimeIndex()` when the timezone was inferred from input dates yet incorrect times were returned when crossing DST boundaries ([GH7835](#), [GH7901](#)).
- Bug in `to_excel()` where a negative sign was being prepended to positive infinity and was absent for negative infinity ([GH7949](#))
- Bug in area plot draws legend with incorrect `alpha` when `stacked=True` ([GH8027](#))
- `Period` and `PeriodIndex` addition/subtraction with `np.timedelta64` results in incorrect internal representations ([GH7740](#))
- Bug in `Holiday` with no offset or observance ([GH7987](#))
- Bug in `DataFrame.to_latex` formatting when columns or index is a `MultiIndex` ([GH7982](#)).
- Bug in `DateOffset` around Daylight Savings Time produces unexpected results ([GH5175](#)).
- Bug in `DataFrame.shift` where empty columns would throw `ZeroDivisionError` on numpy 1.7 ([GH8019](#))
- Bug in installation where `html_encoding/*.html` wasn't installed and therefore some tests were not running correctly ([GH7927](#)).
- Bug in `read_html` where bytes objects were not tested for in `_read` ([GH7927](#)).
- Bug in `DataFrame.stack()` when one of the column levels was a datelike ([GH8039](#))

- Bug in broadcasting numpy scalars with DataFrame ([GH8116](#))
- Bug in pivot\_table performed with nameless index and columns raises KeyError ([GH8103](#))
- Bug in DataFrame.plot(kind='scatter') draws points and errorbars with different colors when the color is specified by c keyword ([GH8081](#))
- Bug in Float64Index where iat and at were not testing and were failing ([GH8092](#)).
- Bug in DataFrame.boxplot() where y-limits were not set correctly when producing multiple axes ([GH7528](#), [GH5517](#)).
- Bug in read\_csv where line comments were not handled correctly given a custom line terminator or delim\_whitespace=True ([GH8122](#)).
- Bug in read\_html where empty tables caused a StopIteration ([GH7575](#))
- Bug in casting when setting a column in a same-dtype block ([GH7704](#))
- Bug in accessing groups from a GroupBy when the original grouper was a tuple ([GH8121](#)).
- Bug in .at that would accept integer indexers on a non-integer index and do fallback ([GH7814](#))
- Bug with kde plot and NaNs ([GH8182](#))
- Bug in GroupBy.count with float32 data type where nan values were not excluded ([GH8169](#)).
- Bug with stacked barplots and NaNs ([GH8175](#)).
- Bug in resample with non evenly divisible offsets (e.g. '7s') ([GH8371](#))
- Bug in interpolation methods with the limit keyword when no values needed interpolating ([GH7173](#)).
- Bug where col\_space was ignored in DataFrame.to\_string() when header=False ([GH8230](#)).
- Bug with DatetimeIndex.asof incorrectly matching partial strings and returning the wrong date ([GH8245](#)).
- Bug in plotting methods modifying the global matplotlib rcParams ([GH8242](#)).
- Bug in DataFrame.\_\_setitem\_\_ that caused errors when setting a dataframe column to a sparse array ([GH8131](#))
- Bug where Dataframe.boxplot() failed when entire column was empty ([GH8181](#)).
- Bug with messed variables in radviz visualization ([GH8199](#)).
- Bug in interpolation methods with the limit keyword when no values needed interpolating ([GH7173](#)).
- Bug where col\_space was ignored in DataFrame.to\_string() when header=False ([GH8230](#)).
- Bug in to\_clipboard that would clip long column data ([GH8305](#))
- Bug in DataFrame terminal display: Setting max\_column/max\_rows to zero did not trigger auto-resizing of dfs to fit terminal width/height ([GH7180](#)).
- Bug in OLS where running with "cluster" and "nw\_lags" parameters did not work correctly, but also did not throw an error ([GH5884](#)).
- Bug in DataFrame.dropna that interpreted non-existent columns in the subset argument as the 'last column' ([GH8303](#))
- Bug in Index.intersection on non-monotonic non-unique indexes ([GH8362](#)).
- Bug in masked series assignment where mismatching types would break alignment ([GH8387](#))
- Bug in NDFrame.equals gives false negatives with dtype=object ([GH8437](#))
- Bug in assignment with indexer where type diversity would break alignment ([GH8258](#))

- Bug in `NDFrame.loc` indexing when row/column names were lost when target was a list/ndarray ([GH6552](#))
- Regression in `NDFrame.loc` indexing when rows/columns were converted to `Float64Index` if target was an empty list/ndarray ([GH7774](#))
- Bug in `Series` that allows it to be indexed by a `DataFrame` which has unexpected results. Such indexing is no longer permitted ([GH8444](#))
- Bug in item assignment of a `DataFrame` with multi-index columns where right-hand-side columns were not aligned ([GH7655](#))
- Suppress `FutureWarning` generated by NumPy when comparing object arrays containing NaN for equality ([GH7065](#))
- Bug in `DataFrame.eval()` where the dtype of the `not` operator (`~`) was not correctly inferred as `bool`.

## 1.23 v0.14.1 (July 11, 2014)

This is a minor release from 0.14.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- Highlights include:
  - New methods `select_dtypes()` to select columns based on the dtype and `sem()` to calculate the standard error of the mean.
  - Support for dateutil timezones (see [docs](#)).
  - Support for ignoring full line comments in the `read_csv()` text parser.
  - New documentation section on *Options and Settings*.
  - Lots of bug fixes.
- *Enhancements*
- *API Changes*
- *Performance Improvements*
- *Experimental Changes*
- *Bug Fixes*

### 1.23.1 API changes

- Openpyxl now raises a `ValueError` on construction of the openpyxl writer instead of warning on pandas import ([GH7284](#)).
- For `StringMethods.extract`, when no match is found, the result - only containing NaN values - now also has `dtype=object` instead of `float` ([GH7242](#))
- `Period` objects no longer raise a `TypeError` when compared using `==` with another object that *isn't* a `Period`. Instead when comparing a `Period` with another object using `==` if the other object isn't a `Period` `False` is returned. ([GH7376](#))



- Previously, the behaviour on resetting the time or not in `offsets.apply`, `rollforward` and `rollback` operations differed between offsets. With the support of the `normalize` keyword for all offsets (see below) with a default value of `False` (preserve time), the behaviour changed for certain offsets (`BusinessMonthBegin`, `MonthEnd`, `BusinessMonthEnd`, `CustomBusinessMonthEnd`, `BusinessYearBegin`, `LastWeekOfMonth`, `FY5253Quarter`, `LastWeekOfMonth`, `Easter`):

```
In [6]: from pandas.tseries import offsets

In [7]: d = pd.Timestamp('2014-01-01 09:00')

# old behaviour < 0.14.1
In [8]: d + offsets.MonthEnd()
Out[8]: Timestamp('2014-01-31 00:00:00')
```

Starting from 0.14.1 all offsets preserve time by default. The old behaviour can be obtained with `normalize=True`

```
# new behaviour
In [1]: d + offsets.MonthEnd()
Out[1]: Timestamp('2014-01-31 09:00:00')

In [2]: d + offsets.MonthEnd(normalize=True)
Out[2]: Timestamp('2014-01-31 00:00:00')
```

Note that for the other offsets the default behaviour did not change.

- Add back `#N/A` `N/A` as a default NA value in text parsing, (regression from 0.12) (GH5521)
- Raise a `TypeError` on inplace-setting with a `.where` and a non `np.nan` value as this is inconsistent with a set-item expression like `df[mask] = None` (GH7656)

## 1.23.2 Enhancements

- Add `dropna` argument to `value_counts` and `nunique` (GH5569).
- Add `select_dtypes()` method to allow selection of columns based on dtype (GH7316). See *the docs*.
- All offsets supports the `normalize` keyword to specify whether `offsets.apply`, `rollforward` and `rollback` resets the time (hour, minute, etc) or not (default `False`, preserves time) (GH7156):

```
In [3]: import pandas.tseries.offsets as offsets

In [4]: day = offsets.Day()

In [5]: day.apply(Timestamp('2014-01-01 09:00'))
Out[5]: Timestamp('2014-01-02 09:00:00')

In [6]: day = offsets.Day(normalize=True)

In [7]: day.apply(Timestamp('2014-01-01 09:00'))
Out[7]: Timestamp('2014-01-02 00:00:00')
```

- `PeriodIndex` is represented as the same format as `DatetimeIndex` (GH7601)
- `StringMethods` now work on empty Series (GH7242)
- The file parsers `read_csv` and `read_table` now ignore line comments provided by the parameter `comment`, which accepts only a single character for the C reader. In particular, they allow for comments before file data begins (GH2685)

- Add `NotImplementedError` for simultaneous use of `chunksize` and `nrows` for `read_csv()` (GH6774).
- Tests for basic reading of public S3 buckets now exist (GH7281).
- `read_html` now sports an `encoding` argument that is passed to the underlying parser library. You can use this to read non-ascii encoded web pages (GH7323).
- `read_excel` now supports reading from URLs in the same way that `read_csv` does. (GH6809)
- Support for `dateutil` timezones, which can now be used in the same way as `pytz` timezones across pandas. (GH4688)

```
In [8]: rng = date_range('3/6/2012 00:00', periods=10, freq='D',
...:                    tz='dateutil/Europe/London')
...:
In [9]: rng.tz
Out[9]: tzfile('/usr/share/zoneinfo/Europe/London')
```

See *the docs*.

- Implemented `sem` (standard error of the mean) operation for `Series`, `DataFrame`, `Panel`, and `Groupby` (GH6897)
- Add `nlargest` and `nsmallest` to the `Series` `groupby` whitelist, which means you can now use these methods on a `SeriesGroupBy` object (GH7053).
- All offsets apply, `rollforward` and `rollback` can now handle `np.datetime64`, previously results in `ApplyTypeError` (GH7452)
- `Period` and `PeriodIndex` can contain `NaT` in its values (GH7485)
- Support pickling `Series`, `DataFrame` and `Panel` objects with non-unique labels along *item* axis (`index`, `columns` and `items` respectively) (GH7370).
- Improved inference of `datetime/timedelta` with mixed null objects. Regression from 0.13.1 in interpretation of an object `Index` with all null elements (GH7431)

### 1.23.3 Performance

- Improvements in dtype inference for numeric operations involving yielding performance gains for dtypes: `int64`, `timedelta64`, `datetime64` (GH7223)
- Improvements in `Series.transform` for significant performance gains (GH6496)
- Improvements in `DataFrame.transform` with `ufuncs` and built-in grouper functions for significant performance gains (GH7383)
- Regression in `groupby` aggregation of `datetime64` dtypes (GH7555)
- Improvements in `MultiIndex.from_product` for large iterables (GH7627)

### 1.23.4 Experimental

- `pandas.io.data.Options` has a new method, `get_all_data` method, and now consistently returns a multi-indexed `DataFrame` (GH5602)
- `io.gbq.read_gbq` and `io.gbq.to_gbq` were refactored to remove the dependency on the Google `bq.py` command line client. This submodule now uses `httplib2` and the Google `apiclient` and `oauth2client` API client libraries which should be more stable and, therefore, reliable than `bq.py`. See *the docs*. (GH6937).

### 1.23.5 Bug Fixes

- Bug in `DataFrame.where` with a symmetric shaped frame and a passed other of a `DataFrame` (GH7506)
- Bug in Panel indexing with a multi-index axis (GH7516)
- Regression in datetimelike slice indexing with a duplicated index and non-exact end-points (GH7523)
- Bug in `setitem` with list-of-lists and single vs mixed types (GH7551:)
- Bug in `timeops` with non-aligned Series (GH7500)
- Bug in `timedelta` inference when assigning an incomplete Series (GH7592)
- Bug in `groupby.nth` with a Series and integer-like column name (GH7559)
- Bug in `Series.get` with a boolean accessor (GH7407)
- Bug in `value_counts` where `NaT` did not qualify as missing (`NaN`) (GH7423)
- Bug in `to_timedelta` that accepted invalid units and misinterpreted 'm/h' (GH7611, GH6423)
- Bug in line plot doesn't set correct `xlim` if `secondary_y=True` (GH7459)
- Bug in grouped `hist` and `scatter` plots use old `figsize` default (GH7394)
- Bug in plotting subplots with `DataFrame.plot`, `hist` clears passed `ax` even if the number of subplots is one (GH7391).
- Bug in plotting subplots with `DataFrame.boxplot` with `by` kw raises `ValueError` if the number of subplots exceeds 1 (GH7391).
- Bug in subplots displays `ticklabels` and `labels` in different rule (GH5897)
- Bug in `Panel.apply` with a multi-index as an axis (GH7469)
- Bug in `DatetimeIndex.insert` doesn't preserve `name` and `tz` (GH7299)
- Bug in `DatetimeIndex.asobject` doesn't preserve `name` (GH7299)
- Bug in multi-index slicing with datetimelike ranges (strings and Timestamps), (GH7429)
- Bug in `Index.min` and `max` doesn't handle `nan` and `NaT` properly (GH7261)
- Bug in `PeriodIndex.min/max` results in `int` (GH7609)
- Bug in `resample` where `fill_method` was ignored if you passed `how` (GH2073)
- Bug in `TimeGrouper` doesn't exclude column specified by `key` (GH7227)
- Bug in `DataFrame` and `Series` `bar` and `barh` plot raises `TypeError` when `bottom` and `left` keyword is specified (GH7226)
- Bug in `DataFrame.hist` raises `TypeError` when it contains non numeric column (GH7277)
- Bug in `Index.delete` does not preserve `name` and `freq` attributes (GH7302)
- Bug in `DataFrame.query()/eval` where local string variables with the `@` sign were being treated as temporaries attempting to be deleted (GH7300).
- Bug in `Float64Index` which didn't allow duplicates (GH7149).
- Bug in `DataFrame.replace()` where truthy values were being replaced (GH7140).
- Bug in `StringMethods.extract()` where a single match group Series would use the matcher's name instead of the group name (GH7313).
- Bug in `isnull()` when `mode.use_inf_as_null == True` where `isnull` wouldn't test `True` when it encountered an `inf/-inf` (GH7315).

- Bug in `inferred_freq` results in `None` for eastern hemisphere timezones (GH7310)
- Bug in `Easter` returns incorrect date when offset is negative (GH7195)
- Bug in broadcasting with `.div`, integer dtypes and divide-by-zero (GH7325)
- Bug in `CustomBusinessDay.apply` raises `NameError` when `np.datetime64` object is passed (GH7196)
- Bug in `MultiIndex.append`, `concat` and `pivot_table` don't preserve timezone (GH6606)
- Bug in `.loc` with a list of indexers on a single-multi index level (that is not nested) (GH7349)
- Bug in `Series.map` when mapping a dict with tuple keys of different lengths (GH7333)
- Bug all `StringMethods` now work on empty `Series` (GH7242)
- Fix delegation of `read_sql` to `read_sql_query` when query does not contain 'select' (GH7324).
- Bug where a string column name assignment to a `DataFrame` with a `Float64Index` raised a `TypeError` during a call to `np.isnan` (GH7366).
- Bug where `NDFrame.replace()` didn't correctly replace objects with `Period` values (GH7379).
- Bug in `.ix` `getitem` should always return a `Series` (GH7150)
- Bug in multi-index slicing with incomplete indexers (GH7399)
- Bug in multi-index slicing with a step in a sliced level (GH7400)
- Bug where negative indexers in `DatetimeIndex` were not correctly sliced (GH7408)
- Bug where `NaT` wasn't repr'd correctly in a `MultiIndex` (GH7406, GH7409).
- Bug where bool objects were converted to `nan` in `convert_objects` (GH7416).
- Bug in `quantile` ignoring the `axis` keyword argument (GH7306)
- Bug where `nanops._maybe_null_out` doesn't work with complex numbers (GH7353)
- Bug in several `nanops` functions when `axis==0` for 1-dimensional `nan` arrays (GH7354)
- Bug where `nanops.nanmedian` doesn't work when `axis==None` (GH7352)
- Bug where `nanops._has_infs` doesn't work with many dtypes (GH7357)
- Bug in `StataReader.data` where reading a 0-observation dta failed (GH7369)
- Bug in `StataReader` when reading Stata 13 (117) files containing fixed width strings (GH7360)
- Bug in `StataWriter` where encoding was ignored (GH7286)
- Bug in `DatetimeIndex` comparison doesn't handle `NaT` properly (GH7529)
- Bug in passing input with `tzinfo` to some offsets `apply`, `rollforward` or `rollback` resets `tzinfo` or raises `ValueError` (GH7465)
- Bug in `DatetimeIndex.to_period`, `PeriodIndex.asobject`, `PeriodIndex.to_timestamp` doesn't preserve name (GH7485)
- Bug in `DatetimeIndex.to_period` and `PeriodIndex.to_timestamp` handle `NaT` incorrectly (GH7228)
- Bug in `offsets.apply`, `rollforward` and `rollback` may return normal datetime (GH7502)
- Bug in `resample` raises `ValueError` when target contains `NaT` (GH7227)
- Bug in `Timestamp.tz_localize` resets nanosecond info (GH7534)
- Bug in `DatetimeIndex.asobject` raises `ValueError` when it contains `NaT` (GH7539)

- Bug in `Timestamp.__new__` doesn't preserve nanosecond properly ([GH7610](#))
- Bug in `Index.astype(float)` where it would return an object dtype Index ([GH7464](#)).
- Bug in `DataFrame.reset_index` loses tz ([GH3950](#))
- Bug in `DatetimeIndex.freqstr` raises `AttributeError` when `freq` is `None` ([GH7606](#))
- Bug in `GroupBy.size` created by `TimeGrouper` raises `AttributeError` ([GH7453](#))
- Bug in single column bar plot is misaligned ([GH7498](#)).
- Bug in area plot with tz-aware time series raises `ValueError` ([GH7471](#))
- Bug in non-monotonic `Index.union` may preserve name incorrectly ([GH7458](#))
- Bug in `DatetimeIndex.intersection` doesn't preserve timezone ([GH4690](#))
- Bug in `rolling_var` where a window larger than the array would raise an error([GH7297](#))
- Bug with last plotted timeseries dictating `xlim` ([GH2960](#))
- Bug with `secondary_y` axis not being considered for timeseries `xlim` ([GH3490](#))
- Bug in `Float64Index` assignment with a non scalar indexer ([GH7586](#))
- Bug in `pandas.core.strings.str_contains` does not properly match in a case insensitive fashion when `regex=False` and `case=False` ([GH7505](#))
- Bug in `expanding_cov`, `expanding_corr`, `rolling_cov`, and `rolling_corr` for two arguments with mismatched index ([GH7512](#))
- Bug in `to_sql` taking the boolean column as text column ([GH7678](#))
- Bug in grouped `hist` doesn't handle `rot` kw and `sharex` kw properly ([GH7234](#))
- Bug in `.loc` performing fallback integer indexing with object dtype indices ([GH7496](#))
- Bug (regression) in `PeriodIndex` constructor when passed `Series` objects ([GH7701](#)).

## 1.24 v0.14.0 (May 31 , 2014)

This is a major release from 0.13.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

- Highlights include:
  - Officially support Python 3.4
  - SQL interfaces updated to use `sqlalchemy`, See [Here](#).
  - Display interface changes, See [Here](#)
  - MultiIndexing Using Slicers, See [Here](#).
  - Ability to join a singly-indexed `DataFrame` with a multi-indexed `DataFrame`, see [Here](#)
  - More consistency in groupby results and more flexible groupby specifications, See [Here](#)
  - Holiday calendars are now supported in `CustomBusinessDay`, see [Here](#)
  - Several improvements in plotting functions, including: hexbin, area and pie plots, see [Here](#).
  - Performance doc section on I/O operations, See [Here](#)
- *Other Enhancements*

- *API Changes*
- *Text Parsing API Changes*
- *Groupby API Changes*
- *Performance Improvements*
- *Prior Deprecations*
- *Deprecations*
- *Known Issues*
- *Bug Fixes*

**Warning:** In 0.14.0 all NDFrame based containers have undergone significant internal refactoring. Before that each block of homogeneous data had its own labels and extra care was necessary to keep those in sync with the parent container's labels. This should not have any visible user/API behavior changes ([GH6745](#))

## 1.24.1 API changes

- `read_excel` uses 0 as the default sheet ([GH6573](#))
- `iloc` will now accept out-of-bounds indexers for slices, e.g. a value that exceeds the length of the object being indexed. These will be excluded. This will make pandas conform more with python/numpy indexing of out-of-bounds values. A single indexer that is out-of-bounds and drops the dimensions of the object will still raise `IndexError` ([GH6296](#), [GH6299](#)). This could result in an empty axis (e.g. an empty `DataFrame` being returned)

```
In [1]: df1 = DataFrame(np.random.randn(5,2), columns=list('AB'))
```

```
In [2]: df1
```

```
Out [2]:
```

```
      A      B
0  1.583584 -0.438313
1 -0.402537 -0.780572
2 -0.141685  0.542241
3  0.370966 -0.251642
4  0.787484  1.666563
```

```
In [3]: df1.iloc[:,2:3]
```

```

////////////////////////////////////
↪
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]
```

```
In [4]: df1.iloc[:,1:3]
```

```

////////////////////////////////////
↪
      B
0 -0.438313
1 -0.780572
2  0.542241
3 -0.251642
4  1.666563
```

(continues on next page)

(continued from previous page)

```
In [5]: df1.iloc[4:6]
//////////
↪
      A      B
4  0.787484  1.666563
```

These are out-of-bounds selections

```
df1.iloc[[4,5,6]]
IndexError: positional indexers are out-of-bounds

df1.iloc[:,4]
IndexError: single positional indexer is out-of-bounds
```

- Slicing with negative start, stop & step values handles corner cases better (GH6531):
  - `df.iloc[: -len(df)]` is now empty
  - `df.iloc[len(df) : -1]` now enumerates all elements in reverse
- The `DataFrame.interpolate()` keyword `downcast` default has been changed from `infer` to `None`. This is to preserve the original dtype unless explicitly requested otherwise (GH6290).
- When converting a dataframe to HTML it used to return *Empty DataFrame*. This special case has been removed, instead a header with the column names is returned (GH6062).
- `Series` and `Index` now internally share more common operations, e.g. `factorize()`, `nunique()`, `value_counts()` are now supported on `Index` types as well. The `Series.weekday` property from is removed from `Series` for API consistency. Using a `DatetimeIndex/PeriodIndex` method on a `Series` will now raise a `TypeError`. (GH4551, GH4056, GH5519, GH6380, GH7206).
- Add `is_month_start`, `is_month_end`, `is_quarter_start`, `is_quarter_end`, `is_year_start`, `is_year_end` accessors for `DatetimeIndex / Timestamp` which return a boolean array of whether the timestamp(s) are at the start/end of the month/quarter/year defined by the frequency of the `DatetimeIndex / Timestamp` (GH4565, GH6998)
- Local variable usage has changed in `pandas.eval()`/`DataFrame.eval()`/`DataFrame.query()` (GH5987). For the *DataFrame* methods, two things have changed
  - Column names are now given precedence over locals
  - Local variables must be referred to explicitly. This means that even if you have a local variable that is *not* a column you must still refer to it with the '@' prefix.
  - You can have an expression like `df.query('@a < a')` with no complaints from pandas about ambiguity of the name `a`.
  - The top-level `pandas.eval()` function does not allow you use the '@' prefix and provides you with an error message telling you so.
  - `NameResolutionError` was removed because it isn't necessary anymore.
- Define and document the order of column vs index names in `query/eval` (GH6676)
- `concat` will now concatenate mixed `Series` and `DataFrames` using the `Series` name or numbering columns as needed (GH2385). See *the docs*
- Slicing and advanced/boolean indexing operations on `Index` classes as well as `Index.delete()` and `Index.drop()` methods will no longer change the type of the resulting index (GH6440, GH7040)

```
In [6]: i = pd.Index([1, 2, 3, 'a', 'b', 'c'])

In [7]: i[[0,1,2]]
Out[7]: Index([1, 2, 3], dtype='object')

In [8]: i.drop(['a', 'b', 'c'])
Out[8]: Index([1, 2, 3], dtype='object')
```

Previously, the above operation would return `Int64Index`. If you'd like to do this manually, use `Index.astype()`

```
In [9]: i[[0,1,2]].astype(np.int_)
Out[9]: Int64Index([1, 2, 3], dtype='int64')
```

- `set_index` no longer converts `MultiIndex`s to an `Index` of tuples. For example, the old behavior returned an `Index` in this case ([GH6459](#)):

```
# Old behavior, casted MultiIndex to an Index
In [10]: tuple_ind
Out[10]: Index([('a', 'c'), ('a', 'd'), ('b', 'c'), ('b', 'd')], dtype='object')

In [11]: df_multi.set_index(tuple_ind)
Out[11]:
      0      1
(a, c) 0.471435 -1.190976
(a, d) 1.432707 -0.312652
(b, c) -0.720589  0.887163
(b, d)  0.859588 -0.636524

# New behavior
In [12]: mi
Out[12]:
MultiIndex(levels=[['a', 'b'], ['c', 'd']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

In [13]: df_multi.set_index(mi)
Out[13]:
      0      1
a c  0.471435 -1.190976
  d  1.432707 -0.312652
b c -0.720589  0.887163
  d  0.859588 -0.636524
```

This also applies when passing multiple indices to `set_index`:

```
# Old output, 2-level MultiIndex of tuples
In [14]: df_multi.set_index([df_multi.index, df_multi.index])
Out[14]:
      0      1
(a, c) (a, c) 0.471435 -1.190976
(a, d) (a, d) 1.432707 -0.312652
(b, c) (b, c) -0.720589  0.887163
(b, d) (b, d)  0.859588 -0.636524
```

(continues on next page)



(continued from previous page)

```
# New output, 4-level MultiIndex
In [15]: df_multi.set_index([df_multi.index, df_multi.index])
Out [15]:
```

		0	1
a	c a c	0.471435	-1.190976
	d a d	1.432707	-0.312652
b	c b c	-0.720589	0.887163
	d b d	0.859588	-0.636524

- `pairwise` keyword was added to the statistical moment functions `rolling_cov`, `rolling_corr`, `ewmcov`, `ewmcorr`, `expanding_cov`, `expanding_corr` to allow the calculation of moving window covariance and correlation matrices (GH4950). See *Computing rolling pairwise covariances and correlations* in the docs.

```
In [1]: df = DataFrame(np.random.randn(10,4), columns=list('ABCD'))

In [4]: covs = pd.rolling_cov(df[['A','B','C']], df[['B','C','D']], 5,
↳ pairwise=True)

In [5]: covs[df.index[-1]]
Out [5]:
```

	B	C	D
A	0.035310	0.326593	-0.505430
B	0.137748	-0.006888	-0.005383
C	-0.006888	0.861040	0.020762

- `Series.iteritems()` is now lazy (returns an iterator rather than a list). This was the documented behavior prior to 0.14. (GH6760)
- Added `nunique` and `value_counts` functions to `Index` for counting unique elements. (GH6734)
- `stack` and `unstack` now raise a `ValueError` when the `level` keyword refers to a non-unique item in the `Index` (previously raised a `KeyError`). (GH6738)
- drop unused `order` argument from `Series.sort`; args now are in the same order as `Series.order`; add `na_position` arg to conform to `Series.order` (GH6847)
- default sorting algorithm for `Series.order` is now `quicksort`, to conform with `Series.sort` (and `numpy` defaults)
- add `inplace` keyword to `Series.order/sort` to make them inverses (GH6859)
- `DataFrame.sort` now places NaNs at the beginning or end of the sort according to the `na_position` parameter. (GH3917)
- accept `TextFileReader` in `concat`, which was affecting a common user idiom (GH6583), this was a regression from 0.13.1
- Added `factorize` functions to `Index` and `Series` to get indexer and unique values (GH7090)
- `describe` on a `DataFrame` with a mix of `Timestamp` and string like objects returns a different `Index` (GH7088). Previously the index was unintentionally sorted.
- Arithmetic operations with **only** `bool` dtypes now give a warning indicating that they are evaluated in Python space for `+`, `-`, and `*` operations and raise for all others (GH7011, GH6762, GH7015, GH7210)

```
x = pd.Series(np.random.rand(10) > 0.5)
y = True
x + y # warning generated: should do x | y instead
```

(continues on next page)

(continued from previous page)

```
x / y # this raises because it doesn't make sense  
NotImplementedError: operator '/' not implemented for bool dtypes
```

- In `HDFStore`, `select_as_multiple` will always raise a `KeyError`, when a key or the selector is not found (GH6177)
- `df['col'] = value` and `df.loc[:, 'col'] = value` are now completely equivalent; previously the `.loc` would not necessarily coerce the dtype of the resultant series (GH6149)
- `dtypes` and `ftypes` now return a series with `dtype=object` on empty containers (GH5740)
- `df.to_csv` will now return a string of the CSV data if neither a target path nor a buffer is provided (GH6061)
- `pd.infer_freq()` will now raise a `TypeError` if given an invalid `Series/Index` type (GH6407, GH6463)
- A tuple passed to `DataFrame.sort_index` will be interpreted as the levels of the index, rather than requiring a list of tuple (GH4370)
- all offset operations now return `Timestamp` types (rather than `datetime`), `Business/Week` frequencies were incorrect (GH4069)
- `to_excel` now converts `np.inf` into a string representation, customizable by the `inf_rep` keyword argument (Excel has no native inf representation) (GH6782)
- Replace `pandas.compat.scipy.scoreatpercentile` with `numpy.percentile` (GH6810)
- `.quantile` on a `datetime[ns]` series now returns `Timestamp` instead of `np.datetime64` objects (GH6810)
- change `AssertionError` to `TypeError` for invalid types passed to `concat` (GH6583)
- Raise a `TypeError` when `DataFrame` is passed an iterator as the `data` argument (GH5357)

### 1.24.2 Display Changes

- The default way of printing large `DataFrames` has changed. `DataFrames` exceeding `max_rows` and/or `max_columns` are now displayed in a centrally truncated view, consistent with the printing of a `pandas.Series` (GH5603).

In previous versions, a `DataFrame` was truncated once the dimension constraints were reached and an ellipse (...) signaled that part of the data was cut off.

```

In [1]: import pandas as pd

In [2]: import numpy as np

In [3]: pd.options.display.max_rows = 6

In [4]: pd.options.display.max_columns = 6

In [5]: index = pd.DatetimeIndex(start='20010101', freq='D', periods=10)

In [6]: pd.DataFrame(np.arange(10*10).reshape((10,10)), index=index)
Out[6]:
      0  1  2  3  4  5 ...
2001-01-01  0  1  2  3  4  5 ...
2001-01-02 10 11 12 13 14 15 ...
2001-01-03 20 21 22 23 24 25 ...
2001-01-04 30 31 32 33 34 35 ...
2001-01-05 40 41 42 43 44 45 ...
2001-01-06 50 51 52 53 54 55 ...
... ..
[10 rows x 10 columns]

```

In the current version, large DataFrames are centrally truncated, showing a preview of head and tail in both dimensions.

```

In [24]: pd.DataFrame(np.arange(10*10).reshape((10,10)), index=index)
Out[24]:
      0  1  2 ...  7  8  9
2001-01-01  0  1  2 ...  7  8  9
2001-01-02 10 11 12 ... 17 18 19
2001-01-03 20 21 22 ... 27 28 29
... ..
2001-01-08 70 71 72 ... 77 78 79
2001-01-09 80 81 82 ... 87 88 89
2001-01-10 90 91 92 ... 97 98 99
[10 rows x 10 columns]

```

- allow option 'truncate' for `display.show_dimensions` to only show the dimensions if the frame is truncated (GH6547).

The default for `display.show_dimensions` will now be `truncate`. This is consistent with how Series display length.

```

In [16]: dfd = pd.DataFrame(np.arange(25).reshape(-1,5), index=[0,1,2,3,4],
↳ columns=[0,1,2,3,4])

# show dimensions since this is truncated
In [17]: with pd.option_context('display.max_rows', 2, 'display.max_columns', 2,
.....:                          'display.show_dimensions', 'truncate'):
.....:     print(dfd)
.....:

```

(continues on next page)

(continued from previous page)

```
[5 rows x 5 columns]
```

```
# will not show dimensions since it is not truncated
```

```
In [18]: with pd.option_context('display.max_rows', 10, 'display.max_columns', 40,
```

```
.....:                               'display.show_dimensions', 'truncate'):
```

```
.....:     print(dfd)
```

```
.....:
```

```
↪ 0  1  2  3  4
```

```
0  0  1  2  3  4
```

```
1  5  6  7  8  9
```

```
2 10 11 12 13 14
```

```
3 15 16 17 18 19
```

```
4 20 21 22 23 24
```

- Regression in the display of a MultiIndexed Series with `display.max_rows` is less than the length of the series ([GH7101](#))
- Fixed a bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the `large_repr` set to 'info' ([GH7105](#))
- The `verbose` keyword in `DataFrame.info()`, which controls whether to shorten the `info` representation, is now `None` by default. This will follow the global setting in `display.max_info_columns`. The global setting can be overridden with `verbose=True` or `verbose=False`.
- Fixed a bug with the `info` repr not honoring the `display.max_info_columns` setting ([GH6939](#))
- Offset/freq info now in `Timestamp.__repr__` ([GH4553](#))

### 1.24.3 Text Parsing API Changes

`read_csv()/read_table()` will now be noisier w.r.t invalid options rather than falling back to the `PythonParser`.

- Raise `ValueError` when `sep` specified with `delim_whitespace=True` in `read_csv()/read_table()` ([GH6607](#))
- Raise `ValueError` when `engine='c'` specified with unsupported options in `read_csv()/read_table()` ([GH6607](#))
- Raise `ValueError` when fallback to python parser causes options to be ignored ([GH6607](#))
- Produce `ParserWarning` on fallback to python parser when no options are ignored ([GH6607](#))
- Translate `sep='\s+'` to `delim_whitespace=True` in `read_csv()/read_table()` if no other C-unsupported options specified ([GH6607](#))

### 1.24.4 Groupby API Changes

More consistent behaviour for some groupby methods:

- `groupby head` and `tail` now act more like `filter` rather than an aggregation:

```
In [19]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])

In [20]: g = df.groupby('A')

In [21]: g.head(1)  # filters DataFrame
Out[21]:
   A  B
0  1  2
2  5  6

In [22]: g.apply(lambda x: x.head(1))  # used to simply fall-through
Out[22]:
   A  B
A
1  0  1  2
5  2  5  6
```

- groupby head and tail respect column selection:

```
In [23]: g[['B']].head(1)
Out[23]:
   B
0  2
2  6
```

- groupby nth now reduces by default; filtering can be achieved by passing `as_index=False`. With an optional `dropna` argument to ignore NaN. See [the docs](#).

#### Reducing

```
In [24]: df = DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])

In [25]: g = df.groupby('A')

In [26]: g.nth(0)
Out[26]:
   B
A
1 NaN
5 6.0

# this is equivalent to g.first()
In [27]: g.nth(0, dropna='any')
Out[27]:
   B
A
1 4.0
5 6.0

# this is equivalent to g.last()
In [28]: g.nth(-1, dropna='any')
Out[28]:
   B
A
1 4.0
5 6.0
```

## Filtering

```
In [29]: gf = df.groupby('A',as_index=False)

In [30]: gf.nth(0)
Out[30]:
   A    B
0  1  NaN
2  5  6.0

In [31]: gf.nth(0, dropna='any')
Out[31]:
   A    B
1  1  4.0
5  5  6.0
```

- `groupby` will now not return the grouped column for non-cython functions ([GH5610](#), [GH5614](#), [GH6732](#)), as its already the index

```
In [32]: df = DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B'])

In [33]: g = df.groupby('A')

In [34]: g.count()
Out[34]:
   B
A
1  1
5  2

In [35]: g.describe()
Out[35]:
   B
count mean      std min 25% 50% 75% max
A
1  1.0  4.0      NaN  4.0  4.0  4.0  4.0  4.0
5  2.0  7.0  1.414214  6.0  6.5  7.0  7.5  8.0
```

- passing `as_index` will leave the grouped column in-place (this is not change in 0.14.0)

```
In [36]: df = DataFrame([[1, np.nan], [1, 4], [5, 6], [5, 8]], columns=['A', 'B'])

In [37]: g = df.groupby('A',as_index=False)

In [38]: g.count()
Out[38]:
   A  B
0  1  1
1  5  2

In [39]: g.describe()
Out[39]:
   A      B
count mean      std min 25% 50% 75% max count mean      std min 25% 50% 75% max
A
1  1.0  4.0      NaN  4.0  4.0  4.0  4.0  4.0  1.0  4.0      NaN  4.0  4.0  4.0  4.0
5  2.0  7.0  1.414214  6.0  6.5  7.0  7.5  8.0  1.0  7.0  1.414214  6.0  6.5  7.0  7.5  8.0
```

(continues on next page)

(continued from previous page)

```

1    2.0  5.0  0.0  5.0  5.0  5.0  5.0  5.0  2.0  7.0  1.414214  6.0  6.5  7.0  7.
↪5    8.0

```

- Allow specification of a more complex groupby via `pd.Grouper`, such as grouping by a Time and a string field simultaneously. See [the docs](#). (GH3794)
- Better propagation/preservation of Series names when performing groupby operations:
  - `SeriesGroupBy.agg` will ensure that the name attribute of the original series is propagated to the result (GH6265).
  - If the function provided to `GroupBy.apply` returns a named series, the name of the series will be kept as the name of the column index of the DataFrame returned by `GroupBy.apply` (GH6124). This facilitates `DataFrame.stack` operations where the name of the column index is used as the name of the inserted column containing the pivoted data.

### 1.24.5 SQL

The SQL reading and writing functions now support more database flavors through SQLAlchemy (GH2717, GH4163, GH5950, GH6292). All databases supported by SQLAlchemy can be used, such as PostgreSQL, MySQL, Oracle, Microsoft SQL server (see documentation of SQLAlchemy on [included dialects](#)).

The functionality of providing DBAPI connection objects will only be supported for sqlite3 in the future. The 'mysql' flavor is deprecated.

The new functions `read_sql_query()` and `read_sql_table()` are introduced. The function `read_sql()` is kept as a convenience wrapper around the other two and will delegate to specific function depending on the provided input (database table name or sql query).

In practice, you have to provide a SQLAlchemy engine to the sql functions. To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For an in-memory sqlite database:

```

In [40]: from sqlalchemy import create_engine

# Create your connection.
In [41]: engine = create_engine('sqlite:///memory:')

```

This engine can then be used to write or read data to/from this database:

```

In [42]: df = pd.DataFrame({'A': [1,2,3], 'B': ['a', 'b', 'c']})

In [43]: df.to_sql('db_table', engine, index=False)

```

You can read data from a database by specifying the table name:

```

In [44]: pd.read_sql_table('db_table', engine)
Out[44]:
   A  B
0  1  a
1  2  b
2  3  c

```

or by specifying a sql query:

```
In [45]: pd.read_sql_query('SELECT * FROM db_table', engine)
Out[45]:
```

	A	B
0	1	a
1	2	b
2	3	c

Some other enhancements to the sql functions include:

- support for writing the index. This can be controlled with the `index` keyword (default is `True`).
- specify the column label to use when writing the index with `index_label`.
- specify string columns to parse as datetimes with the `parse_dates` keyword in `read_sql_query()` and `read_sql_table()`.

**Warning:** Some of the existing functions or function aliases have been deprecated and will be removed in future versions. This includes: `tquery`, `uquery`, `read_frame`, `frame_query`, `write_frame`.

**Warning:** The support for the ‘mysql’ flavor when using DBAPI connection objects has been deprecated. MySQL will be further supported with SQLAlchemy engines ([GH6900](#)).

## 1.24.6 MultiIndexing Using Slicers

In 0.14.0 we added a new way to slice multi-indexed objects. You can slice a multi-index by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see *Selection by Label*, including slices, lists of labels, labels, and boolean indexers.

You can use `slice(None)` to select all the contents of *that* level. You do not need to specify all the *deeper* levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

See *the docs* See also issues ([GH6134](#), [GH4036](#), [GH3057](#), [GH2598](#), [GH5641](#), [GH7106](#))

**Warning:** You should specify all axes in the `.loc` specifier, meaning the indexer for the **index** and for the **columns**. There are some ambiguous cases where the passed indexer could be mis-interpreted as indexing *both* axes, rather than into say the MultiIndex for the rows.

You should do this:

```
df.loc[(slice('A1', 'A3'), .....), :]
```

rather than this:

```
df.loc[(slice('A1', 'A3'), .....)]
```

**Warning:** You will need to make sure that the selection axes are fully lexsorted!



```

In [46]: def mklbl(prefix,n):
.....:     return ["%s%s" % (prefix,i) for i in range(n)]
.....:

In [47]: index = MultiIndex.from_product([mklbl('A',4),
.....:                                     mklbl('B',2),
.....:                                     mklbl('C',4),
.....:                                     mklbl('D',2)])
.....:

In [48]: columns = MultiIndex.from_tuples([('a','foo'),('a','bar'),
.....:                                     ('b','foo'),('b','bah')],
.....:                                     names=['lv10','lv11'])
.....:

In [49]: df = DataFrame(np.arange(len(index)*len(columns)).reshape((len(index),
↪len(columns))),
.....:                    index=index,
.....:                    columns=columns).sort_index().sort_index(axis=1)
.....:

In [50]: df
Out[50]:

```

				a		b	
				bar	foo	bah	foo
lv10							
lv11							
A0	B0	C0	D0	1	0	3	2
			D1	5	4	7	6
		C1	D0	9	8	11	10
			D1	13	12	15	14
		C2	D0	17	16	19	18
			D1	21	20	23	22
		C3	D0	25	24	27	26
...				...	...	...	...
A3	B1	C0	D1	229	228	231	230
		C1	D0	233	232	235	234
			D1	237	236	239	238
		C2	D0	241	240	243	242
			D1	245	244	247	246
		C3	D0	249	248	251	250
			D1	253	252	255	254

```

[64 rows x 4 columns]

```

Basic multi-index slicing using slices, lists, and labels.

```

In [51]: df.loc[(slice('A1','A3'),slice(None), ['C1','C3']),:]
Out[51]:

```

				a		b	
				bar	foo	bah	foo
lv10							
lv11							
A1	B0	C1	D0	73	72	75	74
			D1	77	76	79	78
		C3	D0	89	88	91	90
			D1	93	92	95	94
	B1	C1	D0	105	104	107	106
			D1	109	108	111	110
		C3	D0	121	120	123	122
...				...	...	...	...

(continues on next page)

(continued from previous page)

```

A3 B0 C1 D1 205 204 207 206
      C3 D0 217 216 219 218
      D1 221 220 223 222
      B1 C1 D0 233 232 235 234
      D1 237 236 239 238
      C3 D0 249 248 251 250
      D1 253 252 255 254

[24 rows x 4 columns]

```

You can use a `pd.IndexSlice` to shortcut the creation of these slices

```

In [52]: idx = pd.IndexSlice

In [53]: df.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[53]:
lvl0      a      b
lvl1    foo    foo
A0 B0 C1 D0      8     10
      D1     12     14
      C3 D0     24     26
      D1     28     30
      B1 C1 D0     40     42
      D1     44     46
      C3 D0     56     58
...
A3 B0 C1 D1    204    206
      C3 D0    216    218
      D1    220    222
      B1 C1 D0    232    234
      D1    236    238
      C3 D0    248    250
      D1    252    254

[32 rows x 2 columns]

```

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```

In [54]: df.loc['A1', (slice(None), 'foo')]
Out[54]:
lvl0      a      b
lvl1    foo    foo
B0 C0 D0     64     66
      D1     68     70
      C1 D0     72     74
      D1     76     78
      C2 D0     80     82
      D1     84     86
      C3 D0     88     90
...
B1 C0 D1    100    102
      C1 D0    104    106
      D1    108    110
      C2 D0    112    114
      D1    116    118
      C3 D0    120    122
      D1    124    126

```

(continues on next page)

(continued from previous page)

[16 rows x 2 columns]

**In [55]:** df.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]

////////////////////////////////////

```

↪
lvl0      a      b
lvl1      foo    foo
A0 B0 C1 D0      8     10
          D1     12     14
          C3 D0     24     26
          D1     28     30
        B1 C1 D0     40     42
          D1     44     46
          C3 D0     56     58
...
A3 B0 C1 D1    204    206
          C3 D0    216    218
          D1    220    222
        B1 C1 D0    232    234
          D1    236    238
          C3 D0    248    250
          D1    252    254

```

[32 rows x 2 columns]

Using a boolean indexer you can provide selection related to the *values*.

**In [56]:** mask = df[('a', 'foo')]>200**In [57]:** df.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]**Out [57]:**

```

lvl0      a      b
lvl1      foo    foo
A3 B0 C1 D1    204    206
          C3 D0    216    218
          D1    220    222
        B1 C1 D0    232    234
          D1    236    238
          C3 D0    248    250
          D1    252    254

```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

**In [58]:** df.loc(axis=0)[:, :, ['C1', 'C3']]**Out [58]:**

```

lvl0      a      b
lvl1      bar    foo    bah    foo
A0 B0 C1 D0      9      8     11     10
          D1     13     12     15     14
          C3 D0     25     24     27     26
          D1     29     28     31     30
        B1 C1 D0     41     40     43     42
          D1     45     44     47     46
          C3 D0     57     56     59     58
...
A3 B0 C1 D1    205    204    207    206

```

(continues on next page)

(continued from previous page)

```

      C3 D0  217  216  219  218
          D1  221  220  223  222
B1 C1 D0  233  232  235  234
      D1  237  236  239  238
      C3 D0  249  248  251  250
          D1  253  252  255  254

[32 rows x 4 columns]

```

Furthermore you can *set* the values using these methods

```

In [59]: df2 = df.copy()

In [60]: df2.loc(axis=0)[:,:,['C1','C3']] = -10

In [61]: df2
Out[61]:
lvl0      a      b
lvl1    bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
      D1    5    4    7    6
      C1 D0 -10 -10 -10 -10
      D1 -10 -10 -10 -10
      C2 D0  17  16  19  18
      D1  21  20  23  22
      C3 D0 -10 -10 -10 -10
...
A3 B1 C0 D1  229  228  231  230
      C1 D0 -10 -10 -10 -10
      D1 -10 -10 -10 -10
      C2 D0  241  240  243  242
      D1  245  244  247  246
      C3 D0 -10 -10 -10 -10
      D1 -10 -10 -10 -10

[64 rows x 4 columns]

```

You can use a right-hand-side of an alignable object as well.

```

In [62]: df2 = df.copy()

In [63]: df2.loc[idx[:,:,:,['C1','C3']],:] = df2*1000

In [64]: df2
Out[64]:
lvl0      a      b
lvl1    bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
      D1    5    4    7    6
      C1 D0  9000  8000 11000 10000
      D1 13000 12000 15000 14000
      C2 D0   17   16   19   18
      D1   21   20   23   22
      C3 D0 25000 24000 27000 26000
...
A3 B1 C0 D1   229   228   231   230
      C1 D0 233000 232000 235000 234000

```

(continues on next page)

(continued from previous page)

	D1	237000	236000	239000	238000
C2	D0	241	240	243	242
	D1	245	244	247	246
C3	D0	249000	248000	251000	250000
	D1	253000	252000	255000	254000

[64 rows x 4 columns]

## 1.24.7 Plotting

- Hexagonal bin plots from `DataFrame.plot` with `kind='hexbin'` ([GH5478](#)), See [the docs](#).
- `DataFrame.plot` and `Series.plot` now supports area plot with specifying `kind='area'` ([GH6656](#)), See [the docs](#)
- Pie plots from `Series.plot` and `DataFrame.plot` with `kind='pie'` ([GH6976](#)), See [the docs](#).
- Plotting with Error Bars is now supported in the `.plot` method of `DataFrame` and `Series` objects ([GH3796](#), [GH6834](#)), See [the docs](#).
- `DataFrame.plot` and `Series.plot` now support a `table` keyword for plotting `matplotlib.Table`, See [the docs](#). The `table` keyword can receive the following values.
  - `False`: Do nothing (default).
  - `True`: Draw a table using the `DataFrame` or `Series` called `plot` method. Data will be transposed to meet `matplotlib`'s default layout.
  - `DataFrame` or `Series`: Draw `matplotlib.table` using the passed data. The data will be drawn as displayed in `print` method (not transposed automatically). Also, helper function `pandas.tools.plotting.table` is added to create a table from `DataFrame` and `Series`, and add it to an `matplotlib.Axes`.
- `plot(legend='reverse')` will now reverse the order of legend labels for most plot kinds. ([GH6014](#))
- Line plot and area plot can be stacked by `stacked=True` ([GH6656](#))
- Following keywords are now acceptable for `DataFrame.plot()` with `kind='bar'` and `kind='barh'`:
  - `width`: Specify the bar width. In previous versions, static value 0.5 was passed to `matplotlib` and it cannot be overwritten. ([GH6604](#))
  - `align`: Specify the bar alignment. Default is `center` (different from `matplotlib`). In previous versions, `pandas` passes `align='edge'` to `matplotlib` and adjust the location to `center` by itself, and it results `align` keyword is not applied as expected. ([GH4525](#))
  - `position`: Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1(right/top-end). Default is 0.5 (center). ([GH6604](#))

Because of the default `align` value changes, coordinates of bar plots are now located on integer values (0.0, 1.0, 2.0 ...). This is intended to make bar plot be located on the same coordinates as line plot. However, bar plot may differs unexpectedly when you manually adjust the bar location or drawing area, such as using `set_xlim`, `set_ylim`, etc. In this cases, please modify your script to meet with new coordinates.

- The `parallel_coordinates()` function now takes argument `color` instead of `colors`. A `FutureWarning` is raised to alert that the old `colors` argument will not be supported in a future release. ([GH6956](#))
- The `parallel_coordinates()` and `andrews_curves()` functions now take positional argument `frame` instead of `data`. A `FutureWarning` is raised if the old `data` argument is used by name. ([GH6956](#))

- `DataFrame.boxplot()` now supports `layout` keyword (GH6769)
- `DataFrame.boxplot()` has a new keyword argument, `return_type`. It accepts `'dict'`, `'axes'`, or `'both'`, in which case a namedtuple with the matplotlib axes and a dict of matplotlib Lines is returned.

### 1.24.8 Prior Version Deprecations/Changes

There are prior version deprecations that are taking effect as of 0.14.0.

- Remove `DateRange` in favor of `DatetimeIndex` (GH6816)
- Remove `column` keyword from `DataFrame.sort` (GH4370)
- Remove `precision` keyword from `set_eng_float_format()` (GH395)
- Remove `force_unicode` keyword from `DataFrame.to_string()`, `DataFrame.to_latex()`, and `DataFrame.to_html()`; these function encode in unicode by default (GH2224, GH2225)
- Remove `nanRep` keyword from `DataFrame.to_csv()` and `DataFrame.to_string()` (GH275)
- Remove `unique` keyword from `HDFStore.select_column()` (GH3256)
- Remove `inferTimeRule` keyword from `Timestamp.offset()` (GH391)
- Remove `name` keyword from `get_data_yahoo()` and `get_data_google()` (commit b921d1a)
- Remove `offset` keyword from `DatetimeIndex` constructor (commit 3136390)
- Remove `time_rule` from several rolling-moment statistical functions, such as `rolling_sum()` (GH1042)
- Removed `neg` – boolean operations on numpy arrays in favor of `inv ~`, as this is going to be deprecated in numpy 1.9 (GH6960)

### 1.24.9 Deprecations

- The `pivot_table()/DataFrame.pivot_table()` and `crosstab()` functions now take arguments `index` and `columns` instead of `rows` and `cols`. A `FutureWarning` is raised to alert that the old `rows` and `cols` arguments will not be supported in a future release (GH5505)
- The `DataFrame.drop_duplicates()` and `DataFrame.duplicated()` methods now take argument `subset` instead of `cols` to better align with `DataFrame.dropna()`. A `FutureWarning` is raised to alert that the old `cols` arguments will not be supported in a future release (GH6680)
- The `DataFrame.to_csv()` and `DataFrame.to_excel()` functions now takes argument `columns` instead of `cols`. A `FutureWarning` is raised to alert that the old `cols` arguments will not be supported in a future release (GH6645)
- Indexers will warn `FutureWarning` when used with a scalar indexer and a non-floating point `Index` (GH4892, GH6960)

```
# non-floating point indexes can only be indexed by integers / labels
In [1]: Series(1,np.arange(5))[3.0]
        pandas/core/index.py:469: FutureWarning: scalar indexers for index type_
        ↪Int64Index should be integers and not floating point
Out [1]: 1

In [2]: Series(1,np.arange(5)).iloc[3.0]
        pandas/core/index.py:469: FutureWarning: scalar indexers for index type_
        ↪Int64Index should be integers and not floating point
Out [2]: 1
```

(continues on next page)

(continued from previous page)

```

In [3]: Series(1,np.arange(5)).iloc[3.0:4]
pandas/core/index.py:527: FutureWarning: slice indexers when using iloc_
↪ should be integers and not floating point
Out[3]:
      3      1
dtype: int64

# these are Float64Indexes, so integer or floating point is acceptable
In [4]: Series(1,np.arange(5.))[3]
Out[4]: 1

In [5]: Series(1,np.arange(5.))[3.0]
Out[6]: 1

```

- Numpy 1.9 compat w.r.t. deprecation warnings ([GH6960](#))
- `Panel.shift()` now has a function signature that matches `DataFrame.shift()`. The old positional argument `lags` has been changed to a keyword argument `periods` with a default value of 1. A `FutureWarning` is raised if the old argument `lags` is used by name. ([GH6910](#))
- The order keyword argument of `factorize()` will be removed. ([GH6926](#)).
- Remove the `copy` keyword from `DataFrame.xs()`, `Panel.major_xs()`, `Panel.minor_xs()`. A view will be returned if possible, otherwise a copy will be made. Previously the user could think that `copy=False` would ALWAYS return a view. ([GH6894](#))
- The `parallel_coordinates()` function now takes argument `color` instead of `colors`. A `FutureWarning` is raised to alert that the old `colors` argument will not be supported in a future release. ([GH6956](#))
- The `parallel_coordinates()` and `andrews_curves()` functions now take positional argument `frame` instead of `data`. A `FutureWarning` is raised if the old `data` argument is used by name. ([GH6956](#))
- The support for the 'mysql' flavor when using DBAPI connection objects has been deprecated. MySQL will be further supported with SQLAlchemy engines ([GH6900](#)).
- The following `io.sql` functions have been deprecated: `tquery`, `uquery`, `read_frame`, `frame_query`, `write_frame`.
- The `percentile_width` keyword argument in `describe()` has been deprecated. Use the `percentiles` keyword instead, which takes a list of percentiles to display. The default output is unchanged.
- The default return type of `boxplot()` will change from a dict to a matplotlib Axes in a future release. You can use the future behavior now by passing `return_type='axes'` to `boxplot`.

## 1.24.10 Known Issues

- OpenPyXL 2.0.0 breaks backwards compatibility ([GH7169](#))

## 1.24.11 Enhancements

- `DataFrame` and `Series` will create a `MultiIndex` object if passed a tuples dict, See *the docs* ([GH3323](#))

```
In [65]: Series({'a': 1, ('a', 'a'): 0,
.....:          ('a', 'c'): 2, ('b', 'a'): 3, ('b', 'b'): 4})
.....:

Out[65]:
a b    1
a    0
c    2
b a    3
b    4
dtype: int64

In [66]: DataFrame({'a': {'A': 1, ('A', 'C'): 2,
.....:                    ('a', 'a'): 3, ('A', 'B'): 4,
.....:                    ('a', 'c'): 5, ('A', 'C'): 6,
.....:                    ('b', 'a'): 7, ('A', 'B'): 8,
.....:                    ('b', 'b'): 9, ('A', 'B'): 10}})
.....:

Out[66]:
      a      b
A B  1.0  4.0  5.0  8.0 10.0
C   2.0  3.0  6.0  7.0  NaN
D   NaN  NaN  NaN  NaN  9.0
```

- Added the `sym_diff` method to `Index` ([GH5543](#))
- `DataFrame.to_latex` now takes a `longtable` keyword, which if `True` will return a table in a `longtable` environment. ([GH6617](#))
- Add option to turn off escaping in `DataFrame.to_latex` ([GH6472](#))
- `pd.read_clipboard` will, if the keyword `sep` is unspecified, try to detect data copied from a spreadsheet and parse accordingly. ([GH6223](#))
- Joining a singly-indexed `DataFrame` with a multi-indexed `DataFrame` ([GH3662](#))

See [the docs](#). Joining multi-index DataFrames on both the left and right is not yet supported ATM.

```
In [67]: household = DataFrame(dict(household_id = [1,2,3],
.....:                               male = [0,1,0],
.....:                               wealth = [196087.3,316478.7,294750]),
.....:                               columns = ['household_id', 'male', 'wealth']
.....:                               ).set_index('household_id'))

In [68]: household
Out[68]:
```

	male	wealth
household_id		
1	0	196087.3
2	1	316478.7
3	0	294750.0

```
In [69]: portfolio = DataFrame(dict(household_id = [1,2,2,3,3,3,4],
.....:                               asset_id = ["n10000301109", "n10000289783",
↳ "gb00b03mlx29",
.....:                               "gb00b03mlx29", "lu0197800237",
↳ "n10000289965", np.nan],
.....:                               name = ["ABN Amro", "Robeco", "Royal Dutch Shell
↳ "Royal Dutch Shell",
```

(continues on next page)



(continued from previous page)

```

.....:                                     "AAB Eastern Europe Equity Fund",
↪ "Postbank BioTech Fonds", np.nan],
.....:                                     share = [1.0, 0.4, 0.6, 0.15, 0.6, 0.25, 1.0]),
.....:                                     columns = ['household_id', 'asset_id', 'name', 'share
↪ ']
.....:                                     ).set_index(['household_id', 'asset_id'])
.....:
In [70]: portfolio
Out [70]:

```

household_id	asset_id	name	share
1	nl0000301109	ABN Amro	1.00
2	nl0000289783	Robeco	0.40
	gb00b03mlx29	Royal Dutch Shell	0.60
3	gb00b03mlx29	Royal Dutch Shell	0.15
	lu0197800237	AAB Eastern Europe Equity Fund	0.60
	nl0000289965	Postbank BioTech Fonds	0.25
4	NaN	NaN	1.00

```

In [71]: household.join(portfolio, how='inner')
\\////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↪

```

household_id	asset_id	male	wealth	name	share
1	nl0000301109	0	196087.3	ABN Amro	1.00
2	nl0000289783	1	316478.7	Robeco	0.40
	gb00b03mlx29	1	316478.7	Royal Dutch Shell	0.60
3	gb00b03mlx29	0	294750.0	Royal Dutch Shell	0.15
	lu0197800237	0	294750.0	AAB Eastern Europe Equity Fund	0.60
	nl0000289965	0	294750.0	Postbank BioTech Fonds	0.25

- `quotechar`, `doublequote`, and `escapechar` can now be specified when using `DataFrame.to_csv` (GH5414, GH4528)
- Partially sort by only the specified levels of a MultiIndex with the `sort_remaining` boolean kwarg. (GH3984)
- Added `to_julian_date` to `TimeStamp` and `DatetimeIndex`. The Julian Date is used primarily in astronomy and represents the number of days from noon, January 1, 4713 BC. Because nanoseconds are used to define the time in pandas the actual range of dates that you can use is 1678 AD to 2262 AD. (GH4041)
- `DataFrame.to_stata` will now check data for compatibility with Stata data types and will upcast when needed. When it is not possible to losslessly upcast, a warning is issued (GH6327)
- `DataFrame.to_stata` and `StataWriter` will accept keyword arguments `time_stamp` and `data_label` which allow the time stamp and dataset label to be set when creating a file. (GH6545)
- `pandas.io.gbq` now handles reading unicode strings properly. (GH5940)
- *Holidays Calendars* are now available and can be used with the `CustomBusinessDay` offset (GH6719)
- `Float64Index` is now backed by a `float64` dtype ndarray instead of an object dtype array (GH6471).
- Implemented `Panel.pct_change` (GH6904)
- Added `how` option to rolling-moment functions to dictate how to handle resampling; `rolling_max()` defaults to max, `rolling_min()` defaults to min, and all others default to mean (GH6297)
- `CustomBuisnessMonthBegin` and `CustomBusinessMonthEnd` are now available (GH6866)

- `Series.quantile()` and `DataFrame.quantile()` now accept an array of quantiles.
- `describe()` now accepts an array of percentiles to include in the summary statistics ([GH4196](#))
- `pivot_table` can now accept `Groupby` by `index` and `columns` keywords ([GH6913](#))

```

In [72]: import datetime

In [73]: df = DataFrame({
....:     'Branch' : 'A A A A A B'.split(),
....:     'Buyer' : 'Carl Mark Carl Carl Joe Joe'.split(),
....:     'Quantity' : [1, 3, 5, 1, 8, 1],
....:     'Date' : [datetime.datetime(2013,11,1,13,0), datetime.datetime(2013,9,
↪1,13,5),
....:               datetime.datetime(2013,10,1,20,0), datetime.datetime(2013,10,
↪2,10,0),
....:               datetime.datetime(2013,11,1,20,0), datetime.datetime(2013,10,
↪2,10,0)],
....:     'PayDay' : [datetime.datetime(2013,10,4,0,0), datetime.datetime(2013,
↪10,15,13,5),
....:                 datetime.datetime(2013,9,5,20,0), datetime.datetime(2013,
↪11,2,10,0),
....:                 datetime.datetime(2013,10,7,20,0), datetime.datetime(2013,
↪9,5,10,0)]})
....:

In [74]: df
Out[74]:
   Branch Buyer  Quantity          Date          PayDay
0      A   Carl         1 2013-11-01 13:00:00 2013-10-04 00:00:00
1      A  Mark         3 2013-09-01 13:05:00 2013-10-15 13:05:00
2      A   Carl         5 2013-10-01 20:00:00 2013-09-05 20:00:00
3      A   Carl         1 2013-10-02 10:00:00 2013-11-02 10:00:00
4      A    Joe         8 2013-11-01 20:00:00 2013-10-07 20:00:00
5      B    Joe         1 2013-10-02 10:00:00 2013-09-05 10:00:00

In [75]: pivot_table(df, index=Grouper(freq='M', key='Date'),
....:                 columns=Grouper(freq='M', key='PayDay'),
....:                 values='Quantity', aggfunc=np.sum)
....:
=====
PayDay      2013-09-30  2013-10-31  2013-11-30
Date
2013-09-30          NaN          3.0          NaN
2013-10-31          6.0          NaN          1.0
2013-11-30          NaN          9.0          NaN

```

- Arrays of strings can be wrapped to a specified width (`str.wrap`) (GH6999)
- Add `nsmallest()` and `Series.nlargest()` methods to `Series`, See *the docs* (GH3960)
- `PeriodIndex` fully supports partial string indexing like `DatetimeIndex` (GH7043)

```
In [76]: prng = period_range('2013-01-01 09:00', periods=100, freq='H')
In [77]: ps = Series(np.random.randn(len(prng)), index=prng)
In [78]: ps
```

(continues on next page)

(continued from previous page)

```

Out [78]:
2013-01-01 09:00    0.015696
2013-01-01 10:00   -2.242685
2013-01-01 11:00    1.150036
2013-01-01 12:00    0.991946
2013-01-01 13:00    0.953324
2013-01-01 14:00   -2.021255
2013-01-01 15:00   -0.334077
...
2013-01-05 06:00    0.566534
2013-01-05 07:00    0.503592
2013-01-05 08:00    0.285296
2013-01-05 09:00    0.484288
2013-01-05 10:00    1.363482
2013-01-05 11:00   -0.781105
2013-01-05 12:00   -0.468018
Freq: H, Length: 100, dtype: float64

```

```

In [79]: ps['2013-01-02']
////////////////////////////////////
↪
2013-01-02 00:00    0.553439
2013-01-02 01:00    1.318152
2013-01-02 02:00   -0.469305
2013-01-02 03:00    0.675554
2013-01-02 04:00   -1.817027
2013-01-02 05:00   -0.183109
2013-01-02 06:00    1.058969
...
2013-01-02 17:00    0.076200
2013-01-02 18:00   -0.566446
2013-01-02 19:00    0.036142
2013-01-02 20:00   -2.074978
2013-01-02 21:00    0.247792
2013-01-02 22:00   -0.897157
2013-01-02 23:00   -0.136795
Freq: H, Length: 24, dtype: float64

```

- `read_excel` can now read milliseconds in Excel dates and times with `xlrd >= 0.9.3`. (GH5945)
- `pd.stats.moments.rolling_var` now uses Welford's method for increased numerical stability (GH6817)
- `pd.expanding_apply` and `pd.rolling_apply` now take args and kwargs that are passed on to the func (GH6289)
- `DataFrame.rank()` now has a percentage rank option (GH5971)
- `Series.rank()` now has a percentage rank option (GH5971)
- `Series.rank()` and `DataFrame.rank()` now accept `method='dense'` for ranks without gaps (GH6514)
- Support passing encoding with `xlwt` (GH3710)
- Refactor Block classes removing `Block.items` attributes to avoid duplication in item handling (GH6745, GH6988).
- Testing statements updated to use specialized asserts (GH6175)

### 1.24.12 Performance

- Performance improvement when converting `DatetimeIndex` to floating ordinals using `DatetimeConverter` (GH6636)
- Performance improvement for `DataFrame.shift` (GH5609)
- Performance improvement in indexing into a multi-indexed `Series` (GH5567)
- Performance improvements in single-dtyped indexing (GH6484)
- Improve performance of `DataFrame` construction with certain offsets, by removing faulty caching (e.g. `MonthEnd`, `BusinessMonthEnd`), (GH6479)
- Improve performance of `CustomBusinessDay` (GH6584)
- improve performance of slice indexing on `Series` with string keys (GH6341, GH6372)
- Performance improvement for `DataFrame.from_records` when reading a specified number of rows from an iterable (GH6700)
- Performance improvements in timedelta conversions for integer dtypes (GH6754)
- Improved performance of compatible pickles (GH6899)
- Improve performance in certain reindexing operations by optimizing `take_2d` (GH6749)
- `GroupBy.count()` is now implemented in Cython and is much faster for large numbers of groups (GH7016).

### 1.24.13 Experimental

There are no experimental changes in 0.14.0

### 1.24.14 Bug Fixes

- Bug in `Series ValueError` when index doesn't match data (GH6532)
- Prevent segfault due to `MultiIndex` not being supported in `HDFStore` table format (GH1848)
- Bug in `pd.DataFrame.sort_index` where mergesort wasn't stable when `ascending=False` (GH6399)
- Bug in `pd.tseries.frequencies.to_offset` when argument has leading zeroes (GH6391)
- Bug in version string gen. for dev versions with shallow clones / install from tarball (GH6127)
- Inconsistent tz parsing `Timestamp` / `to_datetime` for current year (GH5958)
- Indexing bugs with reordered indexes (GH6252, GH6254)
- Bug in `.xs` with a `Series` multiindex (GH6258, GH5684)
- Bug in conversion of a string types to a `DatetimeIndex` with a specified frequency (GH6273, GH6274)
- Bug in `eval` where type-promotion failed for large expressions (GH6205)
- Bug in `interpolate` with `inplace=True` (GH6281)
- `HDFStore.remove` now handles start and stop (GH6177)
- `HDFStore.select_as_multiple` handles start and stop the same way as `select` (GH6177)
- `HDFStore.select_as_coordinates` and `select_column` works with a `where` clause that results in filters (GH6177)

- Regression in join of non\_unique\_indexes (GH6329)
- Issue with groupby agg with a single function and a mixed-type frame (GH6337)
- Bug in DataFrame.replace() when passing a non-bool to\_replace argument (GH6332)
- Raise when trying to align on different levels of a multi-index assignment (GH3738)
- Bug in setting complex dtypes via boolean indexing (GH6345)
- Bug in TimeGrouper/resample when presented with a non-monotonic DatetimeIndex that would return invalid results. (GH4161)
- Bug in index name propagation in TimeGrouper/resample (GH4161)
- TimeGrouper has a more compatible API to the rest of the groupers (e.g. groups was missing) (GH3881)
- Bug in multiple grouping with a TimeGrouper depending on target column order (GH6764)
- Bug in pd.eval when parsing strings with possible tokens like ' & ' (GH6351)
- Bug correctly handle placements of -inf in Panels when dividing by integer 0 (GH6178)
- DataFrame.shift with axis=1 was raising (GH6371)
- Disabled clipboard tests until release time (run locally with nosetests -A disabled) (GH6048).
- Bug in DataFrame.replace() when passing a nested dict that contained keys not in the values to be replaced (GH6342)
- str.match ignored the na flag (GH6609).
- Bug in take with duplicate columns that were not consolidated (GH6240)
- Bug in interpolate changing dtypes (GH6290)
- Bug in Series.get, was using a buggy access method (GH6383)
- Bug in hdfstore queries of the form where=[('date', '>=', datetime(2013,1,1)), ('date', '<=', datetime(2014,1,1))] (GH6313)
- Bug in DataFrame.dropna with duplicate indices (GH6355)
- Regression in chained getitem indexing with embedded list-like from 0.12 (GH6394)
- Float64Index with nans not comparing correctly (GH6401)
- eval/query expressions with strings containing the @ character will now work (GH6366).
- Bug in Series.reindex when specifying a method with some nan values was inconsistent (noted on a resample) (GH6418)
- Bug in DataFrame.replace() where nested dicts were erroneously depending on the order of dictionary keys and values (GH5338).
- Perf issue in concatting with empty objects (GH3259)
- Clarify sorting of sym\_diff on Index objects with NaN values (GH6444)
- Regression in MultiIndex.from\_product with a DatetimeIndex as input (GH6439)
- Bug in str.extract when passed a non-default index (GH6348)
- Bug in str.split when passed pat=None and n=1 (GH6466)
- Bug in io.data.DataReader when passed "F-F\_Momentum\_Factor" and data\_source="famafrench" (GH6460)
- Bug in sum of a timedelta64[ns] series (GH6462)

- Bug in `resample` with a timezone and certain offsets (GH6397)
- Bug in `iat/iloc` with duplicate indices on a Series (GH6493)
- Bug in `read_html` where nan's were incorrectly being used to indicate missing values in text. Should use the empty string for consistency with the rest of pandas (GH5129).
- Bug in `read_html` tests where redirected invalid URLs would make one test fail (GH6445).
- Bug in multi-axis indexing using `.loc` on non-unique indices (GH6504)
- Bug that caused `_ref_locs` corruption when slice indexing across columns axis of a DataFrame (GH6525)
- Regression from 0.13 in the treatment of numpy `datetime64` non-ns dtypes in Series creation (GH6529)
- `.names` attribute of MultiIndexes passed to `set_index` are now preserved (GH6459).
- Bug in `setitem` with a duplicate index and an alignable rhs (GH6541)
- Bug in `setitem` with `.loc` on mixed integer Indexes (GH6546)
- Bug in `pd.read_stata` which would use the wrong data types and missing values (GH6327)
- Bug in `DataFrame.to_stata` that lead to data loss in certain cases, and could be exported using the wrong data types and missing values (GH6335)
- `StataWriter` replaces missing values in string columns by empty string (GH6802)
- Inconsistent types in Timestamp addition/subtraction (GH6543)
- Bug in preserving frequency across Timestamp addition/subtraction (GH4547)
- Bug in empty list lookup caused `IndexError` exceptions (GH6536, GH6551)
- `Series.quantile` raising on an object dtype (GH6555)
- Bug in `.xs` with a nan in level when dropped (GH6574)
- Bug in `fillna` with `method='bfill/ffill'` and `datetime64[ns]` dtype (GH6587)
- Bug in sql writing with mixed dtypes possibly leading to data loss (GH6509)
- Bug in `Series.pop` (GH6600)
- Bug in `iloc` indexing when positional indexer matched `Int64Index` of the corresponding axis and no re-ordering happened (GH6612)
- Bug in `fillna` with `limit` and `value` specified
- Bug in `DataFrame.to_stata` when columns have non-string names (GH4558)
- Bug in `compat` with `np.compress`, surfaced in (GH6658)
- Bug in binary operations with a rhs of a Series not aligning (GH6681)
- Bug in `DataFrame.to_stata` which incorrectly handles nan values and ignores `with_index` keyword argument (GH6685)
- Bug in `resample` with extra bins when using an evenly divisible frequency (GH4076)
- Bug in consistency of groupby aggregation when passing a custom function (GH6715)
- Bug in `resample` when `how=None` resample freq is the same as the axis frequency (GH5955)
- Bug in downcasting inference with empty arrays (GH6733)
- Bug in `obj.blocks` on sparse containers dropping all but the last items of same for dtype (GH6748)
- Bug in unpickling `NaT` (`NaTType`) (GH4606)

- Bug in `DataFrame.replace()` where regex metacharacters were being treated as regexs even when `regex=False` ([GH6777](#)).
- Bug in timedelta ops on 32-bit platforms ([GH6808](#))
- Bug in setting a tz-aware index directly via `.index` ([GH6785](#))
- Bug in `expressions.py` where `numexpr` would try to evaluate arithmetic ops ([GH6762](#)).
- Bug in Makefile where it didn't remove Cython generated C files with `make clean` ([GH6768](#))
- Bug with numpy < 1.7.2 when reading long strings from `HDFStore` ([GH6166](#))
- Bug in `DataFrame._reduce` where non bool-like (0/1) integers were being converted into bools. ([GH6806](#))
- Regression from 0.13 with `fillna` and a Series on datetime-like ([GH6344](#))
- Bug in adding `np.timedelta64` to `DatetimeIndex` with timezone outputs incorrect results ([GH6818](#))
- Bug in `DataFrame.replace()` where changing a dtype through replacement would only replace the first occurrence of a value ([GH6689](#))
- Better error message when passing a frequency of 'MS' in `Period` construction ([GH5332](#))
- Bug in `Series.__unicode__` when `max_rows=None` and the Series has more than 1000 rows. ([GH6863](#))
- Bug in `groupby.get_group` where a datetlike wasn't always accepted ([GH5267](#))
- Bug in `groupBy.get_group` created by `TimeGrouper` raises `AttributeError` ([GH6914](#))
- Bug in `DatetimeIndex.tz_localize` and `DatetimeIndex.tz_convert` converting NaT incorrectly ([GH5546](#))
- Bug in arithmetic operations affecting NaT ([GH6873](#))
- Bug in `Series.str.extract` where the resulting Series from a single group match wasn't renamed to the group name
- Bug in `DataFrame.to_csv` where setting `index=False` ignored the header kwarg ([GH6186](#))
- Bug in `DataFrame.plot` and `Series.plot`, where the legend behave inconsistently when plotting to the same axes repeatedly ([GH6678](#))
- Internal tests for patching `__finalize__` / bug in merge not finalizing ([GH6923](#), [GH6927](#))
- accept `TextFileReader` in `concat`, which was affecting a common user idiom ([GH6583](#))
- Bug in C parser with leading whitespace ([GH3374](#))
- Bug in C parser with `delim_whitespace=True` and `\r`-delimited lines
- Bug in python parser with explicit multi-index in row following column header ([GH6893](#))
- Bug in `Series.rank` and `DataFrame.rank` that caused small floats (<1e-13) to all receive the same rank ([GH6886](#))
- Bug in `DataFrame.apply` with functions that used `*args` or `**kwargs` and returned an empty result ([GH6952](#))
- Bug in sum/mean on 32-bit platforms on overflows ([GH6915](#))
- Moved `Panel.shift` to `NDFrame.slice_shift` and fixed to respect multiple dtypes. ([GH6959](#))
- Bug in enabling `subplots=True` in `DataFrame.plot` only has single column raises `TypeError`, and `Series.plot` raises `AttributeError` ([GH6951](#))
- Bug in `DataFrame.plot` draws unnecessary axes when enabling `subplots` and `kind=scatter` ([GH6951](#))

- Bug in `read_csv` from a filesystem with non-utf-8 encoding ([GH6807](#))
- Bug in `iloc` when setting / aligning ([GH6766](#))
- Bug causing `UnicodeEncodeError` when `get_dummies` called with unicode values and a prefix ([GH6885](#))
- Bug in `timeseries-with-frequency` plot cursor display ([GH5453](#))
- Bug surfaced in `groupby.plot` when using a `Float64Index` ([GH7025](#))
- Stopped tests from failing if options data isn't able to be downloaded from Yahoo ([GH7034](#))
- Bug in `parallel_coordinates` and `radviz` where reordering of class column caused possible color/class mismatch ([GH6956](#))
- Bug in `radviz` and `andrews_curves` where multiple values of 'color' were being passed to plotting method ([GH6956](#))
- Bug in `Float64Index.isin()` where containing `nan`s would make indices claim that they contained all the things ([GH7066](#)).
- Bug in `DataFrame.boxplot` where it failed to use the axis passed as the `ax` argument ([GH3578](#))
- Bug in the `XlsxWriter` and `XlwtWriter` implementations that resulted in datetime columns being formatted without the time ([GH7075](#)) were being passed to plotting method
- `read_fwf()` treats `None` in `colspec` like regular python slices. It now reads from the beginning or until the end of the line when `colspec` contains a `None` (previously raised a `TypeError`)
- Bug in cache coherence with chained indexing and slicing; add `_is_view` property to `NDFrame` to correctly predict views; mark `is_copy` on `xs` only if its an actual copy (and not a view) ([GH7084](#))
- Bug in `DatetimeIndex` creation from string ndarray with `dayfirst=True` ([GH5917](#))
- Bug in `MultiIndex.from_arrays` created from `DatetimeIndex` doesn't preserve `freq` and `tz` ([GH7090](#))
- Bug in `unstack` raises `ValueError` when `MultiIndex` contains `PeriodIndex` ([GH4342](#))
- Bug in `boxplot` and `hist` draws unnecessary axes ([GH6769](#))
- Regression in `groupby.nth()` for out-of-bounds indexers ([GH6621](#))
- Bug in `quantile` with datetime values ([GH6965](#))
- Bug in `Dataframe.set_index`, `reindex` and `pivot` don't preserve `DatetimeIndex` and `PeriodIndex` attributes ([GH3950](#), [GH5878](#), [GH6631](#))
- Bug in `MultiIndex.get_level_values` doesn't preserve `DatetimeIndex` and `PeriodIndex` attributes ([GH7092](#))
- Bug in `Groupby` doesn't preserve `tz` ([GH3950](#))
- Bug in `PeriodIndex` partial string slicing ([GH6716](#))
- Bug in the HTML repr of a truncated Series or DataFrame not showing the class name with the `large_repr` set to 'info' ([GH7105](#))
- Bug in `DatetimeIndex` specifying `freq` raises `ValueError` when passed value is too short ([GH7098](#))
- Fixed a bug with the `info` repr not honoring the `display.max_info_columns` setting ([GH6939](#))
- Bug `PeriodIndex` string slicing with out of bounds values ([GH5407](#))
- Fixed a memory error in the hashtable implementation/factorizer on resizing of large tables ([GH7157](#))
- Bug in `isnull` when applied to 0-dimensional object arrays ([GH7176](#))



- Bug in `query/eval` where global constants were not looked up correctly ([GH7178](#))
- Bug in recognizing out-of-bounds positional list indexers with `iloc` and a multi-axis tuple indexer ([GH7189](#))
- Bug in `setitem` with a single value, multi-index and integer indices ([GH7190](#), [GH7218](#))
- Bug in expressions evaluation with reversed ops, showing in series-dataframe ops ([GH7198](#), [GH7192](#))
- Bug in multi-axis indexing with  $> 2$  ndim and a multi-index ([GH7199](#))
- Fix a bug where invalid eval/query operations would blow the stack ([GH5198](#))

## 1.25 v0.13.1 (February 3, 2014)

This is a minor release from 0.13.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Added `infer_datetime_format` keyword to `read_csv/to_datetime` to allow speedups for homogeneously formatted datetimes.
- Will intelligently limit display precision for datetime/timedelta formats.
- Enhanced Panel `apply()` method.
- Suggested tutorials in new *Tutorials* section.
- Our pandas ecosystem is growing, We now feature related projects in a new *Pandas Ecosystem* section.
- Much work has been taking place on improving the docs, and a new *Contributing* section has been added.
- Even though it may only be of interest to devs, we <3 our new CI status page: [ScatterCI](#).

**Warning:** 0.13.1 fixes a bug that was caused by a combination of having numpy  $< 1.8$ , and doing chained assignment on a string-like array. Please review [the docs](#), chained indexing can have unexpected results and should generally be avoided.

This would previously segfault:

```
In [1]: df = DataFrame(dict(A = np.array(['foo', 'bar', 'bah', 'foo', 'bar'])))
In [2]: df['A'].iloc[0] = np.nan
In [3]: df
Out[3]:
   A
0 NaN
1 bar
2 bah
3 foo
4 bar
```

The recommended way to do this type of assignment is:

```
In [4]: df = DataFrame(dict(A = np.array(['foo', 'bar', 'bah', 'foo', 'bar'])))

In [5]: df.loc[0, 'A'] = np.nan

In [6]: df
Out[6]:
   A
0 NaN
1 bar
2 bah
3 foo
4 bar
```

### 1.25.1 Output Formatting Enhancements

- `df.info()` view now display dtype info per column ([GH5682](#))
- `df.info()` now honors the option `max_info_rows`, to disable null counts for large frames ([GH5974](#))

```
In [7]: max_info_rows = pd.get_option('max_info_rows')

In [8]: df = DataFrame(dict(A = np.random.randn(10),
...:                        B = np.random.randn(10),
...:                        C = date_range('20130101', periods=10)))
...:

In [9]: df.iloc[3:6, [0, 2]] = np.nan
```

```
# set to not display the null counts
In [10]: pd.set_option('max_info_rows', 0)

In [11]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
A      float64
B      float64
C      datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 320.0 bytes
```

```
# this is the default (same as in 0.13.0)
In [12]: pd.set_option('max_info_rows', max_info_rows)

In [13]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 3 columns):
A      7 non-null float64
B      10 non-null float64
C      7 non-null datetime64[ns]
dtypes: datetime64[ns](1), float64(2)
memory usage: 320.0 bytes
```

- Add `show_dimensions` display option for the new `DataFrame` repr to control whether the dimensions print.

```

In [14]: df = DataFrame([[1, 2], [3, 4]])

In [15]: pd.set_option('show_dimensions', False)

In [16]: df
Out[16]:
   0  1
0  1  2
1  3  4

In [17]: pd.set_option('show_dimensions', True)

In [18]: df
Out[18]:
   0  1
0  1  2
1  3  4

[2 rows x 2 columns]

```

- The `ArrayFormatter` for datetime and `timedelta64` now intelligently limit precision based on the values in the array ([GH3401](#))

Previously output might look like:

```

   age                today      diff
0 2001-01-01 00:00:00 2013-04-19 00:00:00 4491 days, 00:00:00
1 2004-06-01 00:00:00 2013-04-19 00:00:00 3244 days, 00:00:00

```

Now the output looks like:

```

In [19]: df = DataFrame([ Timestamp('20010101'),
   ....:                  Timestamp('20040601') ], columns=['age'])
   ....:

In [20]: df['today'] = Timestamp('20130419')

In [21]: df['diff'] = df['today']-df['age']

In [22]: df
Out[22]:
   age      today      diff
0 2001-01-01 2013-04-19 4491 days
1 2004-06-01 2013-04-19 3244 days

[2 rows x 3 columns]

```

## 1.25.2 API changes

- Add `-NaN` and `-nan` to the default set of NA values ([GH5952](#)). See *NA Values*.
- Added `Series.str.get_dummies` vectorized string method ([GH6021](#)), to extract dummy/indicator variables for separated string columns:

```

In [23]: s = Series(['a', 'a|b', np.nan, 'a|c'])

```

(continues on next page)



(continued from previous page)

```
b    NaN
Length: 2, dtype: float64
```

Now, when `apply` is called on an empty `DataFrame`: if the `reduce` argument is `True` a `Series` will be returned, if it is `False` a `DataFrame` will be returned, and if it is `None` (the default) the function being applied will be called with an empty series to try and guess the return type.

```
In [35]: empty.apply(applied_func, reduce=True)
Out[35]:
a    NaN
b    NaN
Length: 2, dtype: float64

In [36]: empty.apply(applied_func, reduce=False)
Out[36]:
Empty DataFrame
Columns: [a, b]
Index: []

[0 rows x 2 columns]
```

### 1.25.3 Prior Version Deprecations/Changes

There are no announced changes in 0.13 or prior that are taking effect as of 0.13.1

### 1.25.4 Deprecations

There are no deprecations of prior behavior in 0.13.1

### 1.25.5 Enhancements

- `pd.read_csv` and `pd.to_datetime` learned a new `infer_datetime_format` keyword which greatly improves parsing perf in many cases. Thanks to @lexual for suggesting and @danbirken for rapidly implementing. (GH5490, GH6021)

If `parse_dates` is enabled and this flag is set, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

```
# Try to infer the format for the index column
df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
                 infer_datetime_format=True)
```

- `date_format` and `datetime_format` keywords can now be specified when writing to excel files (GH4133)
- `MultiIndex.from_product` convenience function for creating a `MultiIndex` from the cartesian product of a set of iterables (GH6055):

```
In [32]: shades = ['light', 'dark']

In [33]: colors = ['red', 'green', 'blue']
```

(continues on next page)

(continued from previous page)

```
In [34]: MultiIndex.from_product([shades, colors], names=['shade', 'color'])
Out[34]:
MultiIndex(levels=[['dark', 'light'], ['blue', 'green', 'red']],
            labels=[[1, 1, 1, 0, 0, 0], [2, 1, 0, 2, 1, 0]],
            names=['shade', 'color'])
```

- Panel `apply()` will work on non-ufuncs. See *the docs*.

```
In [35]: import pandas.util.testing as tm

In [36]: panel = tm.makePanel(5)

In [37]: panel
Out[37]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [38]: panel['ItemA']
\\repeated slashes\\
↪
      A      B      C      D
2000-01-03  0.694103  1.893534 -1.735349 -0.850346
2000-01-04  0.678630  0.639633  1.210384  1.176812
2000-01-05  0.239556 -0.962029  0.797435 -0.524336
2000-01-06  0.151227 -2.085266 -0.379811  0.700908
2000-01-07  0.816127  1.930247  0.702562  0.984188

[5 rows x 4 columns]
```

### Specifying an `apply` that operates on a Series (to return a single element)

```
In [39]: panel.apply(lambda x: x.dtype, axis='items')
Out[39]:
```

	A	B	C	D
2000-01-03	float64	float64	float64	float64
2000-01-04	float64	float64	float64	float64
2000-01-05	float64	float64	float64	float64
2000-01-06	float64	float64	float64	float64
2000-01-07	float64	float64	float64	float64

```
[5 rows x 4 columns]
```

A similar reduction type operation

```
In [40]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[40]:
```

	ItemA	ItemB	ItemC
A	2.579643	3.062757	0.379252
B	1.416120	-1.960855	0.923558
C	0.595222	-1.079772	-3.118269
D	1.487226	-0.734611	-1.979310

```
[4 rows x 3 columns]
```

This is equivalent to

```
In [41]: panel.sum('major_axis')
Out[41]:
```

	ItemA	ItemB	ItemC
A	2.579643	3.062757	0.379252
B	1.416120	-1.960855	0.923558
C	0.595222	-1.079772	-3.118269
D	1.487226	-0.734611	-1.979310

```
[4 rows x 3 columns]
```

A transformation operation that returns a Panel, but is computing the z-score across the major\_axis

```
In [42]: result = panel.apply(
.....:     lambda x: (x-x.mean())/x.std(),
.....:     axis='major_axis')
.....:

In [43]: result
Out[43]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [44]: result['ItemA']
////////////////////////////////////
```

	A	B	C	D
2000-01-03	0.595800	0.907552	-1.556260	-1.244875
2000-01-04	0.544058	0.200868	0.915883	0.953747
2000-01-05	-0.924165	-0.701810	0.569325	-0.891290
2000-01-06	-1.219530	-1.334852	-0.418654	0.437589
2000-01-07	1.003837	0.928242	0.489705	0.744830

```
[5 rows x 4 columns]
```

- Panel `apply()` operating on cross-sectional slabs. (GH1148)

```
In [45]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T

In [46]: result = panel.apply(f, axis = ['items','major_axis'])

In [47]: result
Out[47]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC

In [48]: result.loc[:, :, 'ItemA']
////////////////////////////////////
```

	A	B	C	D
2000-01-03	0.331409	1.071034	-0.914540	-0.510587

(continues on next page)

(continued from previous page)

```

2000-01-04 -0.741017 -0.118794  0.383277  0.537212
2000-01-05  0.065042 -0.767353  0.655436  0.069467
2000-01-06  0.027932 -0.569477  0.908202  0.610585
2000-01-07  1.116434  1.133591  0.871287  1.004064

```

```
[5 rows x 4 columns]
```

This is equivalent to the following

```

In [49]: result = Panel(dict([ (ax, f(panel.loc[:, :, ax]))
.....:                        for ax in panel.minor_axis ]))
.....:

```

```
In [50]: result
```

```
Out[50]:
```

```

<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC

```

```
In [51]: result.loc[:, :, 'ItemA']
```

```

////////////////////////////////////
↪
          A          B          C          D
2000-01-03  0.331409  1.071034 -0.914540 -0.510587
2000-01-04 -0.741017 -0.118794  0.383277  0.537212
2000-01-05  0.065042 -0.767353  0.655436  0.069467
2000-01-06  0.027932 -0.569477  0.908202  0.610585
2000-01-07  1.116434  1.133591  0.871287  1.004064

```

```
[5 rows x 4 columns]
```

## 1.25.6 Performance

Performance improvements for 0.13.1

- Series datetime/timedelta binary operations ([GH5801](#))
- DataFrame count/dropna for axis=1
- Series.str.contains now has a *regex=False* keyword which can be faster for plain (non-regex) string patterns. ([GH5879](#))
- Series.str.extract ([GH5944](#))
- dtypes/ftypes methods ([GH5968](#))
- indexing with object dtypes ([GH5968](#))
- DataFrame.apply ([GH6013](#))
- Regression in JSON IO ([GH5765](#))
- Index construction from Series ([GH6150](#))



## 1.25.7 Experimental

There are no experimental changes in 0.13.1

## 1.25.8 Bug Fixes

See [V0.13.1 Bug Fixes](#) for an extensive list of bugs that have been fixed in 0.13.1.

See the [full release notes](#) or issue tracker on GitHub for a complete list of all API changes, Enhancements and Bug Fixes.

## 1.26 v0.13.0 (January 3, 2014)

This is a major release from 0.12.0 and includes a number of API changes, several new features and enhancements along with a large number of bug fixes.

Highlights include:

- support for a new index type `Float64Index`, and other Indexing enhancements
- `HDFStore` has a new string based syntax for query specification
- support for new methods of interpolation
- updated `timedelta` operations
- a new string manipulation method `extract`
- Nanosecond support for Offsets
- `isin` for DataFrames

Several experimental features are added, including:

- new `eval/query` methods for expression evaluation
- support for `msgpack` serialization
- an i/o interface to Google's `BigQuery`

There are several new or updated docs sections including:

- [Comparison with SQL](#), which should be useful for those familiar with SQL but still learning pandas.
- [Comparison with R](#), idiom translations from R to pandas.
- [Enhancing Performance](#), ways to enhance pandas performance with `eval/query`.

**Warning:** In 0.13.0 `Series` has internally been refactored to no longer sub-class `ndarray` but instead subclass `NDFrame`, similar to the rest of the pandas containers. This should be a transparent change with only very limited API implications. See [Internal Refactoring](#)

### 1.26.1 API changes

- `read_excel` now supports an integer in its `sheetname` argument giving the index of the sheet to read in (GH4301).

- Text parser now treats anything that reads like inf (“inf”, “Inf”, “-Inf”, “iNf”, etc.) as infinity. (GH4220, GH4219), affecting `read_table`, `read_csv`, etc.
- pandas now is Python 2/3 compatible without the need for 2to3 thanks to @jtratrner. As a result, pandas now uses iterators more extensively. This also led to the introduction of substantive parts of the Benjamin Peterson’s `six` library into `compat`. (GH4384, GH4375, GH4372)
- `pandas.util.compat` and `pandas.util.py3compat` have been merged into `pandas.compat`. `pandas.compat` now includes many functions allowing 2/3 compatibility. It contains both list and iterator versions of `range`, `filter`, `map` and `zip`, plus other necessary elements for Python 3 compatibility. `lmap`, `lzip`, `lrange` and `lfilter` all produce lists instead of iterators, for compatibility with `numpy`, subscripting and pandas constructors. (GH4384, GH4375, GH4372)
- `Series.get` with negative indexers now returns the same as `[]` (GH4390)
- Changes to how `Index` and `MultiIndex` handle metadata (`levels`, `labels`, and `names`) (GH4039):

```
# previously, you would have set levels or labels directly
index.levels = [[1, 2, 3, 4], [1, 2, 4, 4]]

# now, you use the set_levels or set_labels methods
index = index.set_levels([[1, 2, 3, 4], [1, 2, 4, 4]])

# similarly, for names, you can rename the object
# but setting names is not deprecated
index = index.set_names(["bob", "cranberry"])

# and all methods take an inplace kwarg - but return None
index.set_names(["bob", "cranberry"], inplace=True)
```

- All division with `NDFrame` objects is now *truedivision*, regardless of the future import. This means that operating on pandas objects will by default use *floating point* division, and return a floating point dtype. You can use `//` and `floordiv` to do integer division.

#### Integer division

```
In [3]: arr = np.array([1, 2, 3, 4])

In [4]: arr2 = np.array([5, 3, 2, 1])

In [5]: arr / arr2
Out[5]: array([0, 0, 1, 4])

In [6]: Series(arr) // Series(arr2)
Out[6]:
0    0
1    0
2    1
3    4
dtype: int64
```

#### True Division

```
In [7]: pd.Series(arr) / pd.Series(arr2) # no future import required
Out[7]:
0    0.200000
1    0.666667
2    1.500000
```

(continues on next page)

(continued from previous page)

```
3      4.000000
dtype: float64
```

- Infer and downcast dtype if `downcast='infer'` is passed to `fillna/ffill/bfill` ([GH4604](#))
- `__nonzero__` for all `NDFrame` objects, will now raise a `ValueError`, this reverts back to ([GH1073](#), [GH4633](#)) behavior. See [gotchas](#) for a more detailed discussion.

This prevents doing boolean comparison on *entire* pandas objects, which is inherently ambiguous. These all will raise a `ValueError`.

```
if df:
    ....
df1 and df2
s1 and s2
```

Added the `.bool()` method to `NDFrame` objects to facilitate evaluating of single-element boolean Series:

```
In [1]: Series([True]).bool()
Out[1]: True

In [2]: Series([False]).bool()
Out[2]: False

In [3]: DataFrame([True]).bool()
Out[3]: True

In [4]: DataFrame([False]).bool()
Out[4]: False
```

- All non-Index `NDFrames` (`Series`, `DataFrame`, `Panel`, `Panel4D`, `SparsePanel`, etc.), now support the entire set of arithmetic operators and arithmetic flex methods (`add`, `sub`, `mul`, etc.). `SparsePanel` does not support `pow` or `mod` with non-scalars. ([GH3765](#))
- `Series` and `DataFrame` now have a `mode()` method to calculate the statistical mode(s) by axis/Series. ([GH5367](#))
- Chained assignment will now by default warn if the user is assigning to a copy. This can be changed with the option `mode.chained_assignment`, allowed options are `raise/warn/None`. See [the docs](#).

```
In [5]: dfc = DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})
In [6]: pd.set_option('chained_assignment', 'warn')
```

The following warning / exception will show if this is attempted.

```
In [7]: dfc.loc[0]['A'] = 1111
```

```
Traceback (most recent call last)
...
SettingWithCopyWarning:
  A value is trying to be set on a copy of a slice from a DataFrame.
  Try using .loc[row_index,col_indexer] = value instead
```

Here is the correct method of assignment.

```
In [8]: dfc.loc[0, 'A'] = 11

In [9]: dfc
Out[9]:
```

	A	B
0	11	1
1	bbb	2
2	ccc	3

```
[3 rows x 2 columns]
```

- **Panel.reindex** has the following call signature `Panel.reindex(items=None, major_axis=None, minor_axis=None)` to conform with other NDFrame objects. See *Internal Refactoring* for more information.
- **Series.argmax** and **Series.argmin** are now aliased to **Series.idxmax** and **Series.idxmin**. These return the index of the min or max element respectively. Prior to 0.13.0 these would return the position of the min / max element. (GH6214)

### 1.26.2 Prior Version Deprecations/Changes

These were announced changes in 0.12 or prior that are taking effect as of 0.13.0

- Remove deprecated `Factor` (GH3650)
- Remove deprecated `set_printoptions/reset_printoptions` (GH3046)
- Remove deprecated `_verbose_info` (GH3215)
- Remove deprecated `read_clipboard/to_clipboard/ExcelFile/ExcelWriter` from `pandas.io.parsers` (GH3717) These are available as functions in the main pandas namespace (e.g. `pd.read_clipboard`)
- default for `tupleize_cols` is now `False` for both `to_csv` and `read_csv`. Fair warning in 0.12 (GH3604)
- default for `display.max_seq_len` is now 100 rather than `None`. This activates truncated display (“...”) of long sequences in various places. (GH3391)

### 1.26.3 Deprecations

Deprecated in 0.13.0

- deprecated `iterkv`, which will be removed in a future release (this was an alias of `iteritems` used to bypass 2to3’s changes). (GH4384, GH4375, GH4372)
- deprecated the string method `match`, whose role is now performed more idiomatically by `extract`. In a future release, the default behavior of `match` will change to become analogous to `contains`, which returns a boolean indexer. (Their distinction is strictness: `match` relies on `re.match` while `contains` relies on `re.search`.) In this release, the deprecated behavior is the default, but the new behavior is available through the keyword argument `as_indexer=True`.

### 1.26.4 Indexing API Changes

Prior to 0.13, it was impossible to use a label indexer (`.loc/.ix`) to set a value that was not contained in the index of a particular axis. (GH2578). See *the docs*

In the `Series` case this is effectively an appending operation

```
In [10]: s = Series([1,2,3])
```

```
In [11]: s
```

```
Out[11]:
```

```
0    1
```

```
1    2
```

```
2    3
```

```
Length: 3, dtype: int64
```

```
In [12]: s[5] = 5.
```

```
In [13]: s
```

```
Out[13]:
```

```
0    1.0
```

```
1    2.0
```

```
2    3.0
```

```
5    5.0
```

```
Length: 4, dtype: float64
```

```
In [14]: dfi = DataFrame(np.arange(6).reshape(3,2),
.....:                  columns=['A', 'B'])
.....:
```

```
In [15]: dfi
```

```
Out[15]:
```

```
   A  B
```

```
0  0  1
```

```
1  2  3
```

```
2  4  5
```

```
[3 rows x 2 columns]
```

This would previously `KeyError`

```
In [16]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']
```

```
In [17]: dfi
```

```
Out[17]:
```

```
   A  B  C
```

```
0  0  1  0
```

```
1  2  3  2
```

```
2  4  5  4
```

```
[3 rows x 3 columns]
```

This is like an `append` operation.

```
In [18]: dfi.loc[3] = 5
```

```
In [19]: dfi
```

```
Out[19]:
```

```
   A  B  C
```

```
0  0  1  0
```

```
1  2  3  2
```

```
2  4  5  4
```

```
3  5  5  5
```

(continues on next page)

(continued from previous page)

[4 rows x 3 columns]

A Panel setting operation on an arbitrary axis aligns the input to the Panel

```
In [20]: p = pd.Panel(np.arange(16).reshape(2,4,2),
.....:               items=['Item1','Item2'],
.....:               major_axis=pd.date_range('2001/1/12',periods=4),
.....:               minor_axis=['A','B'],dtype='float64')
.....:
```

```
In [21]: p
```

```
Out[21]:
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to B
```

```
In [22]: p.loc[:, :, 'C'] = Series([30,32],index=p.items)
```

```
In [23]: p
```

```
Out[23]:
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2001-01-12 00:00:00 to 2001-01-15 00:00:00
Minor_axis axis: A to C
```

```
In [24]: p.loc[:, :, 'C']
```

```
////////////////////////////////////
↪
      Item1  Item2
2001-01-12   30.0   32.0
2001-01-13   30.0   32.0
2001-01-14   30.0   32.0
2001-01-15   30.0   32.0

[4 rows x 2 columns]
```

### 1.26.5 Float64Index API Change

- Added a new index type, `Float64Index`. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes `[]`, `ix`, `loc` for scalar indexing and slicing work exactly the same. See [the docs](#), (GH263)

Construction is by default for floating type values.

```
In [25]: index = Index([1.5, 2, 3, 4.5, 5])
```

```
In [26]: index
```

```
Out[26]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')
```

```
In [27]: s = Series(range(5),index=index)
```

```
In [28]: s
```

(continues on next page)

(continued from previous page)

```
Out [28]:
1.5    0
2.0    1
3.0    2
4.5    3
5.0    4
Length: 5, dtype: int64
```

Scalar selection for `[]`, `.ix`, `.loc` will always be label based. An integer will match an equal float index (e.g. 3 is equivalent to 3.0)

```
In [29]: s[3]
Out[29]: 2

In [30]: s.loc[3]
Out[30]: 2
```

The only positional indexing is via `iloc`

```
In [31]: s.iloc[3]
Out[31]: 3
```

A scalar index that is not found will raise `KeyError`

Slicing is ALWAYS on the values of the index, for `[]`, `ix`, `loc` and ALWAYS positional with `iloc`

```
In [32]: s[2:4]
Out[32]:
2.0    1
3.0    2
Length: 2, dtype: int64

In [33]: s.loc[2:4]
Out[33]:
2.0    1
3.0    2
Length: 2, dtype: int64

In [34]: s.iloc[2:4]
Out[34]:
3.0    2
4.5    3
Length: 2, dtype: int64
```

In float indexes, slicing using floats are allowed

```
In [35]: s[2.1:4.6]
Out[35]:
3.0    2
4.5    3
Length: 2, dtype: int64

In [36]: s.loc[2.1:4.6]
Out[36]:
3.0    2
```

(continues on next page)

(continued from previous page)

```
4.5      3
Length: 2, dtype: int64
```

- Indexing on other index types are preserved (and positional fallback for `[]`, `ix`), with the exception, that floating point slicing on indexes on non `Float64Index` will now raise a `TypeError`.

```
In [1]: Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type
↳ (Int64Index)

In [1]: Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type
↳ (Int64Index)
```

Using a scalar float indexer will be deprecated in a future version, but is allowed for now.

```
In [3]: Series(range(5))[3.0]
Out[3]: 3
```

## 1.26.6 HDFStore API Changes

- Query Format Changes. A much more string-like query format is now supported. See [the docs](#).

```
In [37]: path = 'test.h5'

In [38]: dfq = DataFrame(randn(10,4),
.....:                  columns=list('ABCD'),
.....:                  index=date_range('20130101', periods=10))
.....:

In [39]: dfq.to_hdf(path, 'dfq', format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.

```
In [40]: read_hdf(path, 'dfq',
.....:             where="index>Timestamp('20130104') & columns=['A', 'B']")
.....:
Out[40]:
```

	A	B
2013-01-05	1.057633	-0.791489
2013-01-06	1.910759	0.787965
2013-01-07	1.043945	2.107785
2013-01-08	0.749185	-0.675521
2013-01-09	-0.276646	1.924533
2013-01-10	0.226363	-2.078618

[6 rows x 2 columns]

Use an inline column reference

```
In [41]: read_hdf(path, 'dfq',
.....:             where="A>0 or C>0")
.....:
Out[41]:
```

	A	B	C	D
--	---	---	---	---

(continues on next page)



(continued from previous page)

```

2013-01-01 -0.414505 -1.425795  0.209395 -0.592886
2013-01-02 -1.473116 -0.896581  1.104352 -0.431550
2013-01-03 -0.161137  0.889157  0.288377 -1.051539
2013-01-04 -0.319561 -0.619993  0.156998 -0.571455
2013-01-05  1.057633 -0.791489 -0.524627  0.071878
2013-01-06  1.910759  0.787965  0.513082 -0.546416
2013-01-07  1.043945  2.107785  1.459927  1.015405
2013-01-08  0.749185 -0.675521  0.440266  0.688972
2013-01-09 -0.276646  1.924533  0.411204  0.890765
2013-01-10  0.226363 -2.078618 -0.387886 -0.087107

[10 rows x 4 columns]

```

- the format keyword now replaces the table keyword; allowed values are fixed(f) or table(t) the same defaults as prior < 0.13.0 remain, e.g. put implies fixed format and append implies table format. This default format can be set as an option by setting `io.hdf.default_format`.

```

In [42]: path = 'test.h5'

In [43]: df = pd.DataFrame(np.random.randn(10,2))

In [44]: df.to_hdf(path, 'df_table', format='table')

In [45]: df.to_hdf(path, 'df_table2', append=True)

In [46]: df.to_hdf(path, 'df_fixed')

In [47]: with pd.HDFStore(path) as store:
.....:     print(store)
.....:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

```

- Significant table writing performance improvements
- handle a passed Series in table format ([GH4330](#))
- can now serialize a `timedelta64[ns]` dtype in a table ([GH3577](#)), See *the docs*.
- added an `is_open` property to indicate if the underlying file handle is `is_open`; a closed store will now report 'CLOSED' when viewing the store (rather than raising an error) ([GH4409](#))
- a close of a `HDFStore` now will close that instance of the `HDFStore` but will only close the actual file if the ref count (by PyTables) w.r.t. all of the open handles are 0. Essentially you have a local instance of `HDFStore` referenced by a variable. Once you close it, it will report closed. Other references (to the same file) will continue to operate until they themselves are closed. Performing an action on a closed file will raise `ClosedFileError`

```

In [48]: path = 'test.h5'

In [49]: df = DataFrame(randn(10,2))

In [50]: store1 = HDFStore(path)

In [51]: store2 = HDFStore(path)

In [52]: store1.append('df', df)

```

(continues on next page)

(continued from previous page)

```

In [53]: store2.append('df2',df)

In [54]: store1
Out[54]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [55]: store2
Out[55]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [56]: store1.close()

In [57]: store2
Out[57]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

In [58]: store2.close()

In [59]: store2
Out[59]:
<class 'pandas.io.pytables.HDFStore'>
File path: test.h5

```

- removed the `_quiet` attribute, replace by a `DuplicateWarning` if retrieving duplicate rows from a table ([GH4367](#))
- removed the `warn` argument from `open`. Instead a `PossibleDataLossError` exception will be raised if you try to use `mode='w'` with an OPEN file handle ([GH4367](#))
- allow a passed locations array or mask as a `where` condition ([GH4467](#)). See [the docs](#) for an example.
- add the keyword `dropna=True` to `append` to change whether ALL nan rows are not written to the store (default is `True`, ALL nan rows are NOT written), also settable via the option `io.hdf.dropna_table` ([GH4625](#))
- pass thru store creation arguments; can be used to support in-memory stores

### 1.26.7 DataFrame repr Changes

The HTML and plain text representations of `DataFrame` now show a truncated view of the table once it exceeds a certain size, rather than switching to the short info view ([GH4886](#), [GH5550](#)). This makes the representation more consistent as small `DataFrames` get larger.

<b>2010-03-29</b>	13.70	13.88	13.39	13.57	158225000	12.98
<b>2010-03-30</b>	13.55	13.64	13.18	13.28	142055200	12.70
	...	...	...	...	...	...

771 rows x 6 columns

To get the info view, call `DataFrame.info()`. If you prefer the info view as the repr for large DataFrames, you can set this by running `set_option('display.large_repr', 'info')`.

### 1.26.8 Enhancements

- `df.to_clipboard()` learned a new `excel` keyword that let's you paste df data directly into excel (enabled by default). (GH5070).
- `read_html` now raises a `URLError` instead of catching and raising a `ValueError` (GH4303, GH4305)
- Added a test for `read_clipboard()` and `to_clipboard()` (GH4282)
- Clipboard functionality now works with PySide (GH4282)
- Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)
- `to_dict` now takes `records` as a possible outtype. Returns an array of column-keyed dictionaries. (GH4936)
- NaN handling in `get_dummies` (GH4446) with `dummy_na`

```
# previously, nan was erroneously counted as 2 here
# now it is not counted at all
In [60]: get_dummies([1, 2, np.nan])
Out[60]:
   1.0  2.0
0    1    0
1    0    1
2    0    0

[3 rows x 2 columns]

# unless requested
In [61]: get_dummies([1, 2, np.nan], dummy_na=True)
Out[61]:
   1.0  2.0  NaN
0    1    0    0
1    0    1    0
2    0    0    1

[3 rows x 3 columns]
```

- `timedelta64[ns]` operations. See [the docs](#).

**Warning:** Most of these operations require `numpy >= 1.7`

Using the new top-level `to_timedelta`, you can convert a scalar or array from the standard `timedelta` format (produced by `to_csv`) into a `timedelta` type (`np.timedelta64` in nanoseconds).

```
In [62]: to_timedelta('1 days 06:05:01.00003')
Out[62]: Timedelta('1 days 06:05:01.000030')

In [63]: to_timedelta('15.5us')
Out[63]: Timedelta('0 days 00:00:00.000015')

In [64]: to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
Out[64]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT], dtype='timedelta64[ns]', freq=None)

In [65]: to_timedelta(np.arange(5), unit='s')
Out[65]: TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'], dtype='timedelta64[ns]', freq=None)

In [66]: to_timedelta(np.arange(5), unit='d')
Out[66]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype='timedelta64[ns]', freq=None)
```

A Series of dtype `timedelta64[ns]` can now be divided by another `timedelta64[ns]` object, or astyped to yield a `float64` typed Series. This is frequency conversion. See [the docs](#) for the docs.

```
In [67]: from datetime import timedelta

In [68]: td = Series(date_range('20130101', periods=4)) - Series(date_range('20121201', periods=4))

In [69]: td[2] += np.timedelta64(timedelta(minutes=5, seconds=3))

In [70]: td[3] = np.nan

In [71]: td
Out[71]:
0    31 days 00:00:00
1    31 days 00:00:00
2    31 days 00:05:03
3                NaT
Length: 4, dtype: timedelta64[ns]

# to days
In [72]: td / np.timedelta64(1, 'D')
Out[72]:
0    31.000000
1    31.000000
2    31.003507
3           NaN
Length: 4, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [73]: td.astype('timedelta64[D]')
```

```

////////////////////////////////////
↪
0      31.0
1      31.0
2      31.0
3       NaN
Length: 4, dtype: float64
```

```
# to seconds
```

```
In [74]: td / np.timedelta64(1, 's')
```

```

////////////////////////////////////
↪
0      2678400.0
1      2678400.0
2      2678703.0
3           NaN
Length: 4, dtype: float64
```

```
In [75]: td.astype('timedelta64[s]')
```

```

////////////////////////////////////
↪
0      2678400.0
1      2678400.0
2      2678703.0
3           NaN
Length: 4, dtype: float64
```

Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series

```
In [76]: td * -1
```

```
Out[76]:
```

```

0      -31 days +00:00:00
1      -31 days +00:00:00
2      -32 days +23:54:57
3                NaT
Length: 4, dtype: timedelta64[ns]
```

```
In [77]: td * Series([1,2,3,4])
```

```

////////////////////////////////////
↪
0      31 days 00:00:00
1      62 days 00:00:00
2      93 days 00:15:09
3                NaT
Length: 4, dtype: timedelta64[ns]
```

Absolute `DateOffset` objects can act equivalently to `timedeltas`

```
In [78]: from pandas import offsets
```

```
In [79]: td + offsets.Minute(5) + offsets.Milli(5)
```

```
Out[79]:
```

```

0      31 days 00:05:00.005000
1      31 days 00:05:00.005000
2      31 days 00:10:03.005000
```

(continues on next page)

(continued from previous page)

```
3          NaT
Length: 4, dtype: timedelta64[ns]
```

Fillna is now supported for timedeltas

```
In [80]: td.fillna(0)
Out[80]:
0    31 days 00:00:00
1    31 days 00:00:00
2    31 days 00:05:03
3     0 days 00:00:00
Length: 4, dtype: timedelta64[ns]

In [81]: td.fillna(timedelta(days=1,seconds=5))
Out[81]:
0    31 days 00:00:00
1    31 days 00:00:00
2    31 days 00:05:03
3     1 days 00:00:05
Length: 4, dtype: timedelta64[ns]
```

You can do numeric reduction operations on timedeltas.

```
In [82]: td.mean()
Out[82]: Timedelta('31 days 00:01:41')

In [83]: td.quantile(.1)
Out[83]: Timedelta('31 days 00:00:00')
```

- `plot(kind='kde')` now accepts the optional parameters `bw_method` and `ind`, passed to `scipy.stats.gaussian_kde()` (for `scipy >= 0.11.0`) to set the bandwidth, and to `gkde.evaluate()` to specify the indices at which it is evaluated, respectively. See [scipy docs](#). ([GH4298](#))
- `DataFrame` constructor now accepts a numpy masked record array ([GH3478](#))
- The new vectorized string method `extract` return regular expression matches more conveniently.

```
In [84]: Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)')
Out[84]:
0    1
1    2
2   NaN
[3 rows x 1 columns]
```

Elements that do not match return NaN. Extracting a regular expression with more than one group returns a `DataFrame` with one column per group.

```
In [85]: Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)')
Out[85]:
0    a    1
1    b    2
2   NaN  NaN
```

(continues on next page)

(continued from previous page)

```
[3 rows x 2 columns]
```

Elements that do not match return a row of NaN. Thus, a Series of messy strings can be *converted* into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects.

Named groups like

```
In [86]: Series(['a1', 'b2', 'c3']).str.extract(
.....:         '(?P<letter>[ab])(?P<digit>\d)')
.....:
Out [86]:
   letter digit
0      a      1
1      b      2
2    NaN    NaN

[3 rows x 2 columns]
```

and optional groups can also be used.

```
In [87]: Series(['a1', 'b2', '3']).str.extract(
.....:         '(?P<letter>[ab])?(?P<digit>\d)')
.....:
Out [87]:
   letter digit
0      a      1
1      b      2
2    NaN      3

[3 rows x 2 columns]
```

- `read_stata` now accepts Stata 13 format ([GH4291](#))
- `read_fwf` now infers the column specifications from the first 100 rows of the file if the data has correctly separated and properly aligned columns using the delimiter provided to the function ([GH4488](#)).
- support for nanosecond times as an offset

**Warning:** These operations require `numpy >= 1.7`

Period conversions in the range of seconds and below were reworked and extended up to nanoseconds. Periods in the nanosecond range are now available.

```
In [88]: date_range('2013-01-01', periods=5, freq='5N')
Out [88]:
DatetimeIndex([
    '2013-01-01 00:00:00',
    '2013-01-01 00:00:00.000000005',
    '2013-01-01 00:00:00.000000010',
    '2013-01-01 00:00:00.000000015',
    '2013-01-01 00:00:00.000000020'],
              dtype='datetime64[ns]', freq='5N')
```

or with frequency as offset

```
In [90]: t = Timestamp('20130101 09:01:02')

In [91]: t + pd.tseries.offsets.Nano(123)
Out[91]: Timestamp('2013-01-01 09:01:02.000000123')
```

- To get the rows where any of the conditions are met:

```
In [92]: dfi = DataFrame({'A': [1, 2, 3, 4], 'B': ['a', 'b', 'f', 'n']})

In [93]: dfi
Out[93]:
   A  B
0  1  a
1  2  b
2  3  f
3  4  n

[4 rows x 2 columns]

In [94]: other = DataFrame({'A': [1, 3, 3, 7], 'B': ['e', 'f', 'f', 'e']})

In [95]: mask = dfi.isin(other)

In [96]: mask
Out[96]:
      A      B
0  True  False
1  False False
2  True   True
3  False False

[4 rows x 2 columns]

In [97]: dfi[mask.any(1)]
////////////////////////////////////
↪
   A  B
0  1  a
2  3  f

[2 rows x 2 columns]
```

- `Series` now supports a `to_frame` method to convert it to a single-column `DataFrame` (GH5164)
- All R datasets listed here <http://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html> can now be



loaded into Pandas objects

```
# note that pandas.rpy was deprecated in v0.16.0
import pandas.rpy.common as com
com.load_data('Titanic')
```

- `tz_localize` can infer a fall daylight savings transition based on the structure of the unlocalized data (GH4230), see [the docs](#)
- `DatetimeIndex` is now in the API documentation, see [the docs](#)
- `json_normalize()` is a new method to allow you to create a flat table from semi-structured JSON data. See [the docs](#) (GH1067)
- Added PySide support for the `qt pandas DataFrameModel` and `DataFrameWidget`.
- Python csv parser now supports `usecols` (GH4335)
- Frequencies gained several new offsets:
  - `LastWeekOfMonth` (GH4637)
  - `FY5253`, and `FY5253Quarter` (GH4511)
- `DataFrame` has a new `interpolate` method, similar to `Series` (GH4434, GH1892)

```
In [98]: df = DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
.....:                  'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})
.....:

In [99]: df.interpolate()
Out[99]:
```

	A	B
0	1.0	0.25
1	2.1	1.50
2	3.4	2.75
3	4.7	4.00
4	5.6	12.20
5	6.8	14.40

```
[6 rows x 2 columns]
```

Additionally, the method argument to `interpolate` has been expanded to include `'nearest'`, `'zero'`, `'slinear'`, `'quadratic'`, `'cubic'`, `'barycentric'`, `'krogh'`, `'piecewise_polynomial'`, `'pchip'`, `'polynomial'`, `'spline'`. The new methods require `scipy`. Consult the [Scipy reference guide](#) and [documentation](#) for more information about when the various methods are appropriate. See [the docs](#).

`Interpolate` now also accepts a `limit` keyword argument. This works similar to `fillna`'s `limit`:

```
In [100]: ser = Series([1, 3, np.nan, np.nan, np.nan, 11])

In [101]: ser.interpolate(limit=2)
Out[101]:
```

0	1.0
1	3.0
2	5.0
3	7.0
4	NaN
5	11.0

```
Length: 6, dtype: float64
```

- Added `wide_to_long` panel data convenience function. See *the docs*.

```
In [102]: np.random.seed(123)

In [103]: df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
.....:                      "A1980" : {0 : "d", 1 : "e", 2 : "f"},
.....:                      "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
.....:                      "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
.....:                      "X"      : dict(zip(range(3), np.random.randn(3)))
.....:                      })

In [104]: df["id"] = df.index

In [105]: df
Out[105]:
```

	A1970	A1980	B1970	B1980	X	id
0	a	d	2.5	3.2	-1.085631	0
1	b	e	1.2	1.3	0.997345	1
2	c	f	0.7	0.1	0.282978	2

```
[3 rows x 6 columns]

In [106]: wide_to_long(df, ["A", "B"], i="id", j="year")
////////////////////////////////////
```

		X	A	B
id	year			
0	1970	-1.085631	a	2.5
1	1970	0.997345	b	1.2
2	1970	0.282978	c	0.7
0	1980	-1.085631	d	3.2
1	1980	0.997345	e	1.3
2	1980	0.282978	f	0.1

```
[6 rows x 3 columns]
```

- `to_csv` now takes a `date_format` keyword argument that specifies how output datetime objects should be formatted. Datetimes encountered in the index, columns, and values will all have this formatting applied. ([GH4313](#))
- `DataFrame.plot` will scatter plot x versus y by passing `kind='scatter'` ([GH2215](#))
- Added support for Google Analytics v3 API segment IDs that also supports v2 IDs. ([GH5271](#))

### 1.26.9 Experimental

- The new `eval()` function implements expression evaluation using `numexpr` behind the scenes. This results in large speedups for complicated expressions involving large DataFrames/Series. For example,

```
In [107]: nrows, ncols = 20000, 100
In [108]: df1, df2, df3, df4 = [DataFrame(randn(nrows, ncols))
.....:                             for _ in range(4)]
.....:
```

```
# eval with NumExpr backend
In [109]: %timeit pd.eval('df1 + df2 + df3 + df4')
8.07 ms +- 212 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
# pure Python evaluation
In [110]: %timeit df1 + df2 + df3 + df4
11.4 ms +- 752 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

For more details, see the [the docs](#)

- Similar to `pandas.eval`, `DataFrame` has a new `DataFrame.eval` method that evaluates an expression in the context of the `DataFrame`. For example,

```
In [111]: df = DataFrame(randn(10, 2), columns=['a', 'b'])

In [112]: df.eval('a + b')
Out[112]:
0    -0.685204
1     1.589745
2     0.325441
3    -1.784153
4    -0.432893
5     0.171850
6     1.895919
7     3.065587
8    -0.092759
9     1.391365
Length: 10, dtype: float64
```

- `query()` method has been added that allows you to select elements of a `DataFrame` using a natural query syntax nearly identical to Python syntax. For example,

```
In [113]: n = 20

In [114]: df = DataFrame(np.random.randint(n, size=(n, 3)), columns=['a', 'b', 'c
↪'])

In [115]: df.query('a < b < c')
Out[115]:
   a  b  c
11  1  5  8
15  8 16 19

[2 rows x 3 columns]
```

selects all the rows of `df` where `a < b < c` evaluates to `True`. For more details see the [the docs](#).

- `pd.read_msgpack()` and `pd.to_msgpack()` are now a supported method of serialization of arbitrary pandas (and python objects) in a lightweight portable binary format. See [the docs](#)

**Warning:** Since this is an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

```
In [116]: df = DataFrame(np.random.rand(5, 2), columns=list('AB'))
```

(continues on next page)

(continued from previous page)

```

In [117]: df.to_msgpack('foo.msg')

In [118]: pd.read_msgpack('foo.msg')
Out[118]:
      A      B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

[5 rows x 2 columns]

In [119]: s = Series(np.random.rand(5), index=date_range('20130101', periods=5))

In [120]: pd.to_msgpack('foo.msg', df, s)

In [121]: pd.read_msgpack('foo.msg')
Out[121]:
[
      A      B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

[5 rows x 2 columns], 2013-01-01    0.022321
2013-01-02    0.227025
2013-01-03    0.383282
2013-01-04    0.193225
2013-01-05    0.110977
Freq: D, Length: 5, dtype: float64]

```

You can pass `iterator=True` to iterator over the unpacked results

```

In [122]: for o in pd.read_msgpack('foo.msg', iterator=True):
.....:     print(o)
.....:
      A      B
0  0.251082  0.017357
1  0.347915  0.929879
2  0.546233  0.203368
3  0.064942  0.031722
4  0.355309  0.524575

[5 rows x 2 columns]
2013-01-01    0.022321
2013-01-02    0.227025
2013-01-03    0.383282
2013-01-04    0.193225
2013-01-05    0.110977
Freq: D, Length: 5, dtype: float64

```

- `pandas.io.gbq` provides a simple way to extract from, and load data into, Google's BigQuery Data Sets by way of pandas DataFrames. BigQuery is a high performance SQL-like database service, useful for performing ad-hoc queries against extremely large datasets. *See the docs*

```

from pandas.io import gbq

# A query to select the average monthly temperatures in the
# in the year 2000 across the USA. The dataset,
# publicdata:samples.gsod, is available on all BigQuery accounts,
# and is based on NOAA gsod data.

query = """SELECT station_number as STATION,
month as MONTH, AVG(mean_temp) as MEAN_TEMP
FROM publicdata:samples.gsod
WHERE YEAR = 2000
GROUP BY STATION, MONTH
ORDER BY STATION, MONTH ASC"""

# Fetch the result set for this query

# Your Google BigQuery Project ID
# To find this, see your dashboard:
# https://console.developers.google.com/iam-admin/projects?authuser=0
projectid = xxxxxxxxx;

df = gbq.read_gbq(query, project_id = projectid)

# Use pandas to process and reshape the dataset

df2 = df.pivot(index='STATION', columns='MONTH', values='MEAN_TEMP')
df3 = pandas.concat([df2.min(), df2.mean(), df2.max()],
                    axis=1,keys=["Min Tem", "Mean Temp", "Max Temp"])

```

The resulting DataFrame is:

```

> df3
      Min Tem  Mean Temp  Max Temp
MONTH
1    -53.336667  39.827892  89.770968
2    -49.837500  43.685219  93.437932
3    -77.926087  48.708355  96.099998
4    -82.892858  55.070087  97.317240
5    -92.378261  61.428117  102.042856
6    -77.703334  65.858888  102.900000
7    -87.821428  68.169663  106.510714
8    -89.431999  68.614215  105.500000
9    -86.611112  63.436935  107.142856
10   -78.209677  56.880838  92.103333
11   -50.125000  48.861228  94.996428
12   -50.332258  42.286879  94.396774

```

**Warning:** To use this module, you will need a BigQuery account. See <<https://cloud.google.com/products/big-query>> for details.

As of 10/10/13, there is a bug in Google's API preventing result sets from being larger than 100,000 rows. A patch is scheduled for the week of 10/14/13.

### 1.26.10 Internal Refactoring

In 0.13.0 there is a major refactor primarily to subclass `Series` from `NDFrame`, which is the base class currently for `DataFrame` and `Panel`, to unify methods and behaviors. `Series` formerly subclassed directly from `ndarray`. (GH4080, GH3862, GH816)

**Warning:** There are two potential incompatibilities from < 0.13.0

- Using certain numpy functions would previously return a Series if passed a Series as an argument. This seems only to affect `np.ones_like`, `np.empty_like`, `np.diff` and `np.where`. These now return `ndarrays`.

```
In [123]: s = Series([1, 2, 3, 4])
```

## Numpy Usage

[illegible]

## Pandonic Usage

```
In [127]: Series(1, index=s.index)
Out[127]:
0    1
1    1
2    1
3    1
Length: 4, dtype: int64

In [128]: s.diff()
Out[128]:
0    NaN
1    1.0
2    1.0
3    1.0
Length: 4, dtype: float64

In [129]: s.where(s>1)
Out[129]:
0    NaN
1    2.0
2    3.0
3    4.0
Length: 4, dtype: float64
```

- Passing a `Series` directly to a cython function expecting an `ndarray` type will no longer work directly, you must pass `Series.values`, See [Enhancing Performance](#)
- `Series(0.5)` would previously return the scalar `0.5`, instead this will return a 1-element `Series`
- This change breaks `rpy2<=2.3.8`. An Issue has been opened against `rpy2` and a workaround is detailed in [GH5698](#). Thanks @JanSchulz.

- Pickle compatibility is preserved for pickles created prior to 0.13. These must be unpickled with `pd.read_pickle`, see [Pickling](#).
- Refactor of `series.py/frame.py/panel.py` to move common code to `generic.py`
  - added `__setup_axes` to created generic NDFrame structures
  - moved methods
    - \* `from_axes, __wrap_array, axes, ix, loc, iloc, shape, empty, swapaxes, transpose, pop`
    - \* `__iter__, keys, __contains__, __len__, __neg__, __invert__`
    - \* `convert_objects, as_blocks, as_matrix, values`
    - \* `__getstate__, __setstate__` (compat remains in frame/panel)
    - \* `__getattr__, __setattr__`
    - \* `_indexed_same, reindex_like, align, where, mask`
    - \* `fillna, replace` (Series `replace` is now consistent with DataFrame)
    - \* `filter` (also added axis argument to selectively filter on a different axis)
    - \* `reindex, reindex_axis, take`
    - \* `truncate` (moved to become part of NDFrame)
- These are API changes which make Panel more consistent with DataFrame
  - `swapaxes` on a Panel with the same axes specified now return a copy
  - support attribute access for setting
  - `filter` supports the same API as the original DataFrame filter
- Reindex called with no arguments will now return a copy of the input object
- `TimeSeries` is now an alias for `Series`. the property `is_time_series` can be used to distinguish (if desired)
- Refactor of Sparse objects to use BlockManager
  - Created a new block type in internals, `SparseBlock`, which can hold multi-dtypes and is non-consolidatable. `SparseSeries` and `SparseDataFrame` now inherit more methods from there hierarchy (Series/DataFrame), and no longer inherit from `SparseArray` (which instead is the object of the `SparseBlock`)
  - Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
  - Operations on sparse structures within DataFrames should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
  - enable `setitem` on `SparseSeries` for boolean/integer/slices
  - `SparsePanels` implementation is unchanged (e.g. not using BlockManager, needs work)
- added `ftypes` method to Series/DataFrame, similar to `dtypes`, but indicates if the underlying is sparse/dense (as well as the dtype)
- All NDFrame objects can now use `__finalize__()` to specify various values to propagate to new objects from an existing one (e.g. name in Series will follow more automatically now)
- Internal type checking is now done via a suite of generated classes, allowing `isinstance(value, klass)` without having to directly import the klass, courtesy of @jtratrner

- Bug in Series update where the parent frame is not updating its cache based on changes ([GH4080](#)) or types ([GH3217](#)), fillna ([GH3386](#))
- Indexing with dtype conversions fixed ([GH4463](#), [GH4204](#))
- Refactor Series.reindex to core/generic.py ([GH4604](#), [GH4618](#)), allow method= in reindexing on a Series to work
- Series.copy no longer accepts the order parameter and is now consistent with NDFrame copy
- Refactor rename methods to core/generic.py; fixes Series.rename for ([GH4605](#)), and adds rename with the same signature for Panel
- Refactor clip methods to core/generic.py ([GH4798](#))
- Refactor of \_get\_numeric\_data/\_get\_bool\_data to core/generic.py, allowing Series/Panel functionality
- Series (for index) / Panel (for items) now allow attribute access to its elements ([GH1903](#))

```
In [130]: s = Series([1,2,3],index=list('abc'))

In [131]: s.b
Out[131]: 2

In [132]: s.a = 5

In [133]: s
Out[133]:
a    5
b    2
c    3
Length: 3, dtype: int64
```

### 1.26.11 Bug Fixes

See *V0.13.0 Bug Fixes* for an extensive list of bugs that have been fixed in 0.13.0.

See the *full release notes* or issue tracker on GitHub for a complete list of all API changes, Enhancements and Bug Fixes.

## 1.27 v0.12.0 (July 24, 2013)

This is a major release from 0.11.0 and includes several new features and enhancements along with a large number of bug fixes.

Highlights include a consistent I/O API naming scheme, routines to read html, write multi-indexes to csv files, read & write STATA data files, read & write JSON format files, Python 3 support for HDFStore, filtering of groupby expressions via `filter`, and a revamped `replace` routine that accepts regular expressions.

### 1.27.1 API changes

- The I/O API is now much more consistent with a set of top level reader functions accessed like `pd.read_csv()` that generally return a pandas object.
  - `read_csv`



```

- read_excel
- read_hdf
- read_sql
- read_json
- read_html
- read_stata
- read_clipboard

```

The corresponding writer functions are object methods that are accessed like `df.to_csv()`

```

- to_csv
- to_excel
- to_hdf
- to_sql
- to_json
- to_html
- to_stata
- to_clipboard

```

- Fix modulo and integer division on Series, DataFrames to act similarly to float dtypes to return `np.nan` or `np.inf` as appropriate (GH3590). This correct a numpy bug that treats integer and float dtypes differently.

```
In [1]: p = DataFrame({ 'first' : [4,5,8], 'second' : [0,0,3] })
```

```
In [2]: p % 0
```

```
Out[2]:
```

```

      first  second
0      NaN      NaN
1      NaN      NaN
2      NaN      NaN

[3 rows x 2 columns]

```

```
In [3]: p % p
```

```

////////////////////////////////////
↪
      first  second
0      0.0      NaN
1      0.0      NaN
2      0.0      0.0

[3 rows x 2 columns]

```

```
In [4]: p / p
```

```

////////////////////////////////////
↪
      first  second
0      1.0      NaN
1      1.0      NaN
2      1.0      1.0

```

(continues on next page)

(continued from previous page)

[3 rows x 2 columns]

**In [5]:** p / 0

```

////////////////////////////////////

```

```

↪
   first  second
0    inf    NaN
1    inf    NaN
2    inf    inf

```

[3 rows x 2 columns]

- Add `squeeze` keyword to `groupby` to allow reduction from `DataFrame` -> `Series` if groups are unique. This is a Regression from 0.10.1. We are reverting back to the prior behavior. This means `groupby` will return the same shaped objects whether the groups are unique or not. Revert this issue ([GH2893](#)) with ([GH3596](#)).

```

In [6]: df2 = DataFrame([{"val1": 1, "val2": 20}, {"val1":1, "val2": 19},
...:                      {"val1":1, "val2": 27}, {"val1":1, "val2": 12}])
...:

```

```

In [7]: def func(dataf):
...:     return dataf["val2"] - dataf["val2"].mean()
...:

```

```

# squeezing the result frame to a series (because we have unique groups)

```

```

In [8]: df2.groupby("val1", squeeze=True).apply(func)

```

**Out [8]:**

```

0    0.5
1   -0.5
2    7.5
3   -7.5

```

```

Name: 1, Length: 4, dtype: float64

```

```

# no squeezing (the default, and behavior in 0.10.1)

```

```

In [9]: df2.groupby("val1").apply(func)

```

```

////////////////////////////////////Out [9]:

```

```

↪
val2    0    1    2    3
val1
1      0.5 -0.5  7.5 -7.5

```

[1 rows x 4 columns]

- Raise on `iloc` when boolean indexing with a label based indexer mask e.g. a boolean `Series`, even with integer labels, will raise. Since `iloc` is purely positional based, the labels on the `Series` are not alignable ([GH3631](#))

This case is rarely used, and there are plenty of alternatives. This preserves the `iloc` API to be *purely* positional based.

```

In [10]: df = DataFrame(lrange(5), list('ABCDE'), columns=['a'])

```

```

In [11]: mask = (df.a%2 == 0)

```

```

In [12]: mask

```

**Out [12]:**

```

A      True

```

(continues on next page)

(continued from previous page)

```

B    False
C     True
D    False
E     True
Name: a, Length: 5, dtype: bool

```

```
# this is what you should use
```

```
In [13]: df.loc[mask]
```

```

=====
↪
a
A  0
C  2
E  4

```

```
[3 rows x 1 columns]
```

```
# this will work as well
```

```
In [14]: df.iloc[mask.values]
```

```

=====
↪
a
A  0
C  2
E  4

```

```
[3 rows x 1 columns]
```

`df.iloc[mask]` will raise a `ValueError`

- The `raise_on_error` argument to plotting functions is removed. Instead, plotting functions raise a `TypeError` when the dtype of the object is object to remind you to avoid object arrays whenever possible and thus you should cast to an appropriate numeric dtype if you need to plot something.
- Add `colormap` keyword to `DataFrame` plotting methods. Accepts either a matplotlib colormap object (ie, `matplotlib.cm.jet`) or a string name of such an object (ie, `'jet'`). The colormap is sampled to select the color for each column. Please see [Colormaps](#) for more information. (GH3860)
- `DataFrame.interpolate()` is now deprecated. Please use `DataFrame.fillna()` and `DataFrame.replace()` instead. (GH3582, GH3675, GH3676)
- the method and axis arguments of `DataFrame.replace()` are deprecated
- `DataFrame.replace` 's `infer_types` parameter is removed and now performs conversion by default. (GH3907)
- Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if True, default is False (same as prior to 0.12) (GH3679)
- Implement `__nonzero__` for `NDFrame` objects (GH3691, GH3696)
- IO api
  - added top-level function `read_excel` to replace the following, The original API is deprecated and will be removed in a future version

```

from pandas.io.parsers import ExcelFile
xls = ExcelFile('path_to_file.xls')
xls.parse('Sheet1', index_col=None, na_values=['NA'])

```

With

```
import pandas as pd
pd.read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

- added top-level function `read_sql` that is equivalent to the following

```
from pandas.io.sql import read_frame
read_frame(...)
```

- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)
- Do not allow astypes on `datetime64[ns]` except to object, and `timedelta64[ns]` to object/int (GH3425)
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations (GH3726). The following operations now raise a `TypeError` when performed on a `Series` and return an *empty* `Series` when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of slice objects:
  - `sum`, `prod`, `mean`, `std`, `var`, `skew`, `kurt`, `corr`, and `cov`
- `read_html` now defaults to `None` when reading, and falls back on `bs4` + `html5lib` when `lxml` fails to parse. a list of parsers to try until success is also valid
- The internal pandas class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the baseclass for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). (GH4090, GH4092)
- New `StringMixin` that, given a `__unicode__` method, gets python 2 and python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. (GH4090, GH4092)

### 1.27.2 I/O Enhancements

- `pd.read_html()` can now parse HTML strings, files or urls and return `DataFrames`, courtesy of @cpcloud. (GH3477, GH3605, GH3606, GH3616). It works with a *single* parser backend: `BeautifulSoup4` + `html5lib` *See the docs*

You can use `pd.read_html()` to read the output from `DataFrame.to_html()` like so

```
In [15]: df = DataFrame({'a': range(3), 'b': list('abc')})

In [16]: print(df)
   a  b
0  0  a
1  1  b
2  2  c

[3 rows x 2 columns]

In [17]: html = df.to_html()

In [18]: alist = pd.read_html(html, index_col=0)

In [19]: print(df == alist[0])
   a  b
```

(continues on next page)

(continued from previous page)

```

0  True  True
1  True  True
2  True  True

[3 rows x 2 columns]
```

Note that `alist` here is a Python list so `pd.read_html()` and `DataFrame.to_html()` are not inverses.

- `pd.read_html()` no longer performs hard conversion of date strings ([GH3656](#)).

**Warning:** You may have to install an older version of BeautifulSoup4, [See the installation docs](#)

- Added module for reading and writing Stata files: `pandas.io.stata` ([GH1512](#)) accessible via `read_stata` top-level function for reading, and `to_stata` `DataFrame` method for writing, [See the docs](#)
- Added module for reading and writing json format files: `pandas.io.json` accessible via `read_json` top-level function for reading, and `to_json` `DataFrame` method for writing, [See the docs](#) various issues ([GH1226](#), [GH3804](#), [GH3876](#), [GH3867](#), [GH1305](#))
- `MultiIndex` column support for reading and writing csv format files
  - The header option in `read_csv` now accepts a list of the rows from which to read the index.
  - The option, `tupleize_cols` can now be specified in both `to_csv` and `read_csv`, to provide compatibility for the pre 0.12 behavior of writing and reading `MultiIndex` columns via a list of tuples. The default in 0.12 is to write lists of tuples and *not* interpret list of tuples as a `MultiIndex` column.

Note: The default behavior in 0.12 remains unchanged from prior versions, but starting with 0.13, the default to write and read `MultiIndex` columns will be in the new format. ([GH3571](#), [GH1651](#), [GH3141](#))

- If an `index_col` is not specified (e.g. you don't have an index, or wrote it with `df.to_csv(..., index=False)`), then any names on the columns index will be *lost*.

```

In [20]: from pandas.util.testing import makeCustomDataframe as mkdf

In [21]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)

In [22]: df.to_csv('mi.csv')

In [23]: print(open('mi.csv').read())
C0,,C_10_g0,C_10_g1,C_10_g2
C1,,C_11_g0,C_11_g1,C_11_g2
C2,,C_12_g0,C_12_g1,C_12_g2
C3,,C_13_g0,C_13_g1,C_13_g2
R0,R1,,,
R_10_g0,R_11_g0,R0C0,R0C1,R0C2
R_10_g1,R_11_g1,R1C0,R1C1,R1C2
R_10_g2,R_11_g2,R2C0,R2C1,R2C2
R_10_g3,R_11_g3,R3C0,R3C1,R3C2
R_10_g4,R_11_g4,R4C0,R4C1,R4C2

In [24]: pd.read_csv('mi.csv', header=[0,1,2,3], index_col=[0,1])
\\
→
C0          C_10_g0 C_10_g1 C_10_g2
```

(continues on next page)

(continued from previous page)

```

C1          C_11_g0 C_11_g1 C_11_g2
C2          C_12_g0 C_12_g1 C_12_g2
C3          C_13_g0 C_13_g1 C_13_g2
R0      R1
R_10_g0 R_11_g0      R0C0      R0C1      R0C2
R_10_g1 R_11_g1      R1C0      R1C1      R1C2
R_10_g2 R_11_g2      R2C0      R2C1      R2C2
R_10_g3 R_11_g3      R3C0      R3C1      R3C2
R_10_g4 R_11_g4      R4C0      R4C1      R4C2

[5 rows x 3 columns]

```

- Support for HDFStore (via PyTables 3.0.0) on Python3
- Iterator support via `read_hdf` that automatically opens and closes the store when iteration is finished. This is only for *tables*

```

In [25]: path = 'store_iterator.h5'

In [26]: DataFrame(randn(10,2)).to_hdf(path, 'df', table=True)

In [27]: for df in read_hdf(path, 'df', chunksize=3):
....:     print df
....:
      0      1
0  0.713216 -0.778461
1 -0.661062  0.862877
2  0.344342  0.149565
      0      1
3 -0.626968 -0.875772
4 -0.930687 -0.218983
5  0.949965 -0.442354
      0      1
6 -0.402985  1.111358
7 -0.241527 -0.670477
8  0.049355  0.632633
      0      1
9 -1.502767 -1.225492

```

- `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters

### 1.27.3 Other Enhancements

- `DataFrame.replace()` now allows regular expressions on contained `Series` with object dtype. See the examples section in the regular docs [Replacing via String Expression](#)

For example you can do

```

In [25]: df = DataFrame({'a': list('ab..'), 'b': [1, 2, 3, 4]})

In [26]: df.replace(regex=r'\s*\.\s*', value=np.nan)
Out[26]:
   a  b
0  a  1
1  b  2

```

(continues on next page)

(continued from previous page)

```
2 NaN 3
3 NaN 4

[4 rows x 2 columns]
```

to replace all occurrences of the string ' .' with zero or more instances of surrounding whitespace with NaN.

Regular string replacement still works as expected. For example, you can do

```
In [27]: df.replace(' .', np.nan)
Out[27]:
   a  b
0  a  1
1  b  2
2 NaN 3
3 NaN 4

[4 rows x 2 columns]
```

to replace all occurrences of the string ' .' with NaN.

- `pd.melt()` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame.
- `pd.set_option()` now allows N option, value pairs ([GH3667](#)).

Let's say that we had an option 'a.b' and another option 'b.c'. We can set them at the same time:

```
In [28]: pd.get_option('a.b')
Out[28]: 2

In [29]: pd.get_option('b.c')
Out[29]: 3

In [30]: pd.set_option('a.b', 1, 'b.c', 4)

In [31]: pd.get_option('a.b')
Out[31]: 1

In [32]: pd.get_option('b.c')
Out[32]: 4
```

- The `filter` method for group objects returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```
In [33]: sf = Series([1, 1, 2, 3, 3, 3])

In [34]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
Out[34]:
3    3
4    3
5    3
Length: 3, dtype: int64
```

The argument of `filter` must a function that, applied to the group as a whole, returns True or False.

Another useful operation is filtering out elements that belong to groups with only a couple members.

```
In [35]: dff = DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [36]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[36]:
```

	A	B
2	2	b
3	3	b
4	4	b
5	5	b

```
[4 rows x 2 columns]
```

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [37]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[37]:
```

	A	B
0	NaN	NaN
1	NaN	NaN
2	2.0	b
3	3.0	b
4	4.0	b
5	5.0	b
6	NaN	NaN
7	NaN	NaN

```
[8 rows x 2 columns]
```

- Series and DataFrame hist methods now take a `figsize` argument ([GH3834](#))
- DatetimeIndexes no longer try to convert mixed-integer indexes during join operations ([GH3877](#))
- Timestamp.min and Timestamp.max now represent valid Timestamp instances instead of the default date-time.min and datetime.max (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and BeautifulSoup==4.2.0 is detected ([GH4214](#))

## 1.27.4 Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. ([GH2301](#))

---

**Note:** This uses the `numpy.busdaycalendar` API introduced in Numpy 1.7 and therefore requires Numpy 1.7.0 or newer.

---

```
In [38]: from pandas.tseries.offsets import CustomBusinessDay

In [39]: from datetime import datetime

# As an interesting example, let's look at Egypt where
# a Friday-Saturday weekend is observed.
In [40]: weekmask_egypt = 'Sun Mon Tue Wed Thu'

# They also observe International Workers' Day so let's
```

(continues on next page)



(continued from previous page)

```
# add that for a couple of years
In [41]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01
↳')]

In [42]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_
↳egypt)

In [43]: dt = datetime(2013, 4, 30)

In [44]: print(dt + 2 * bday_egypt)
2013-05-05 00:00:00

In [45]: dts = date_range(dt, periods=5, freq=bday_egypt)

In [46]: print(Series(dts.weekday, dts).map(Series('Mon Tue Wed Thu Fri Sat Sun'.
↳split()))))
2013-04-30    Tue
2013-05-02    Thu
2013-05-05    Sun
2013-05-06    Mon
2013-05-07    Tue
Freq: C, Length: 5, dtype: object
```

### 1.27.5 Bug Fixes

- Plotting functions now raise a `TypeError` before trying to plot anything if the associated objects have a `dtype` of `object` (GH1818, GH3572, GH3911, GH3912), but they will try to convert object arrays to numeric arrays if possible so that you can still plot, for example, an object array with floats. This happens before any drawing takes place which eliminates any spurious plots from showing up.
- `fillna` methods now raise a `TypeError` if the `value` parameter is a list or tuple.
- `Series.str` now supports iteration (GH3638). You can iterate over the individual elements of each string in the `Series`. Each iteration yields a `Series` with either a single character at each index of the original `Series` or `NaN`. For example,

```
In [47]: strs = 'go', 'bow', 'joe', 'slow'

In [48]: ds = Series(strs)

In [49]: for s in ds.str:
.....:     print(s)
.....:
0    g
1    b
2    j
3    s
Length: 4, dtype: object
0    o
1    o
2    o
3    l
Length: 4, dtype: object
0    NaN
1    w
```

(continues on next page)

(continued from previous page)

```

2      e
3      o
Length: 4, dtype: object
0      NaN
1      NaN
2      NaN
3      w
Length: 4, dtype: object

```

```
In [50]: s
```

```

////////////////////////////////////
↪
0      NaN
1      NaN
2      NaN
3      w
Length: 4, dtype: object

```

```
In [51]: s.dropna().values.item() == 'w'
```

```

////////////////////////////////////
↪ True

```

The last element yielded by the iterator will be a `Series` containing the last element of the longest string in the `Series` with all other elements being `NaN`. Here since `'slow'` is the longest string and there are no other strings with the same length `'w'` is the only non-null string in the yielded `Series`.

- `HDFStore`
  - will retain index attributes (`freq,tz,name`) on recreation ([GH3499](#))
  - will warn with a `AttributeConflictWarning` if you are attempting to append an index with a different frequency than the existing, or attempting to append an index with a different name than the existing
  - support datelike columns with a timezone as `data_columns` ([GH2852](#))
- Non-unique index support clarified ([GH3468](#)).
  - Fix assigning a new index to a duplicate index in a `DataFrame` would fail ([GH3468](#))
  - Fix construction of a `DataFrame` with a duplicate index
  - `ref_locs` support to allow duplicative indices across dtypes, allows `iget` support to always find the index (even across dtypes) ([GH2194](#))
  - `applymap` on a `DataFrame` with a non-unique index now works (removed warning) ([GH2786](#)), and fix ([GH3230](#))
  - Fix `to_csv` to handle non-unique columns ([GH3495](#))
  - Duplicate indexes with `getitem` will return items in the correct order ([GH3455](#), [GH3457](#)) and handle missing elements like unique indices ([GH3561](#))
  - Duplicate indexes with an empty `DataFrame.from_records` will return a correct frame ([GH3562](#))
  - Concat to produce a non-unique columns when duplicates are across dtypes is fixed ([GH3602](#))
  - Allow insert/delete to non-unique columns ([GH3679](#))
  - Non-unique indexing with a slice via `loc` and friends fixed ([GH3659](#))
  - Allow insert/delete to non-unique columns ([GH3679](#))

- Extend `reindex` to correctly deal with non-unique indices (GH3679)
- `DataFrame.itertuples()` now works with frames with duplicate column names (GH3873)
- Bug in non-unique indexing via `iloc` (GH4017); added `takeable` argument to `reindex` for location-based taking
- Allow non-unique indexing in series via `.ix/.loc` and `__getitem__` (GH4246)
- Fixed non-unique indexing memory allocation issue with `.ix/.loc` (GH4280)
- `DataFrame.from_records` did not accept empty recarrays (GH3682)
- `read_html` now correctly skips tests (GH3741)
- Fixed a bug where `DataFrame.replace` with a compiled regular expression in the `to_replace` argument wasn't working (GH3907)
- Improved `network` test decorator to catch `IOError` (and therefore `URLError` as well). Added `with_connectivity_check` decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new `optional_args` decorator factory for decorators. (GH3910, GH3914)
- Fixed testing issue where too many sockets were open thus leading to a connection reset issue (GH3982, GH3985, GH4028, GH4054)
- Fixed failing tests in `test_yahoo`, `test_google` where symbols were not retrieved but were being accessed (GH3982, GH3985, GH4028, GH4054)
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN `DataFrame` would barf on a 1xN mask (GH4071)
- Fixed running of `tox` under python3 where the `pickle` import was getting rewritten in an incompatible way (GH4062, GH4063)
- Fixed bug where `sharex` and `sharey` were not being passed to `grouped_hist` (GH4089)
- Fixed bug in `DataFrame.replace` where a nested dict wasn't being iterated over when `regex=False` (GH4115)
- Fixed bug in the parsing of microseconds when using the `format` argument in `to_datetime` (GH4152)
- Fixed bug in `PandasAutoDateLocator` where `invert_xaxis` triggered incorrectly `MilliSecondLocator` (GH3990)
- Fixed bug in plotting that wasn't raising on invalid colormap for matplotlib 1.1.1 (GH4215)
- Fixed the legend displaying in `DataFrame.plot(kind='kde')` (GH4216)
- Fixed bug where Index slices weren't carrying the name attribute (GH4226)
- Fixed bug in initializing `DatetimeIndex` with an array of strings in a certain time zone (GH4229)
- Fixed bug where `html5lib` wasn't being properly skipped (GH4265)
- Fixed bug where `get_data_famafrench` wasn't using the correct file edges (GH4281)

See the [full release notes](#) or issue tracker on GitHub for a complete list.

## 1.28 v0.11.0 (April 22, 2013)

This is a major release from 0.10.1 and includes many new features and enhancements along with a large number of bug fixes. The methods of Selecting Data have had quite a number of additions, and Dtype support is now full-fledged. There are also a number of important API changes that long-time pandas users should pay close attention to.

There is a new section in the documentation, *10 Minutes to Pandas*, primarily geared to new users.

There is a new section in the documentation, *Cookbook*, a collection of useful recipes in pandas (and that we want contributions!).

There are several libraries that are now *Recommended Dependencies*

## 1.28.1 Selection Choices

Starting in 0.11.0, object selection has had a number of user-requested additions in order to support more explicit location based indexing. Pandas now supports three types of multi-axis indexing.

- `.loc` is strictly label based, will raise `KeyError` when the items are not found, allowed inputs are:
  - A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index. This use is **not** an integer position along the index)
  - A list or array of labels ['a', 'b', 'c']
  - A slice object with labels 'a': 'f', (note that contrary to usual python slices, **both** the start and the stop are included!)
  - A boolean array

See more at *Selection by Label*

- `.iloc` is strictly integer position based (from 0 to `length-1` of the axis), will raise `IndexError` when the requested indicies are out of bounds. Allowed inputs are:
  - An integer e.g. 5
  - A list or array of integers [4, 3, 0]
  - A slice object with ints 1:7
  - A boolean array

See more at *Selection by Position*

- `.ix` supports mixed integer and label based access. It is primarily label based, but will fallback to integer positional access. `.ix` is the most general and will support any of the inputs to `.loc` and `.iloc`, as well as support for floating point label schemes. `.ix` is especially useful when dealing with mixed positional and label based hierarchial indexes.

As using integer slices with `.ix` have different behavior depending on whether the slice is interpreted as position based or label based, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at *Advanced Indexing* and *Advanced Hierarchical*.

## 1.28.2 Selection Deprecations

Starting in version 0.11.0, these methods *may* be deprecated in future versions.

- `irow`
- `icol`
- `iget_value`

See the section *Selection by Position* for substitutes.

### 1.28.3 Dtypes

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`, then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [1]: df1 = DataFrame(randn(8, 1), columns = ['A'], dtype = 'float32')
```

```
In [2]: df1
```

```
Out[2]:
      A
0  1.392665
1 -0.123497
2 -0.402761
3 -0.246604
4 -0.288433
5 -0.763434
6  2.069526
7 -1.203569

[8 rows x 1 columns]
```

```
In [3]: df1.dtypes
```

```

////////////////////////////////////
↪
A      float32
Length: 1, dtype: object
```

```
In [4]: df2 = DataFrame(dict( A = Series(randn(8), dtype='float16'),
...:                          B = Series(randn(8)),
...:                          C = Series(randn(8), dtype='uint8') ))
...:
```

```
In [5]: df2
```

```
Out[5]:
      A      B      C
0  0.591797 -0.038605    0
1  0.841309 -0.460478    1
2 -0.500977 -0.310458    0
3 -0.816406  0.866493  254
4 -0.207031  0.245972    0
5 -0.664062  0.319442    1
6  0.580566  1.378512    1
7 -0.965820  0.292502  255

[8 rows x 3 columns]
```

```
In [6]: df2.dtypes
```

```

////////////////////////////////////
↪
A      float16
B      float64
C        uint8
Length: 3, dtype: object
```

```
# here you get some upcasting
```

```
In [7]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2
```

(continues on next page)

(continued from previous page)

```
In [8]: df3
Out[8]:
```

	A	B	C
0	1.984462	-0.038605	0.0
1	0.717812	-0.460478	1.0
2	-0.903737	-0.310458	0.0
3	-1.063011	0.866493	254.0
4	-0.495465	0.245972	0.0
5	-1.427497	0.319442	1.0
6	2.650092	1.378512	1.0
7	-2.169390	0.292502	255.0

```
[8 rows x 3 columns]
```

```
In [9]: df3.dtypes
```

```

A    float32
B    float64
C    float64
Length: 3, dtype: object
```

## 1.28.4 Dtype Conversion

This is lower-common-denominator upcasting, meaning you get the dtype which can accommodate all of the types

```
In [10]: df3.values.dtype
Out[10]: dtype('float64')
```

### Conversion

```
In [11]: df3.astype('float32').dtypes
Out[11]:
```

A	float32
B	float32
C	float32

```
Length: 3, dtype: object
```

### Mixed Conversion

```
In [12]: df3['D'] = '1.'
In [13]: df3['E'] = '1'
In [14]: df3.convert_objects(convert_numeric=True).dtypes
Out[14]:
```

A	float32
B	float64
C	float64
D	float64
E	int64

```
Length: 5, dtype: object

# same, but specific dtype conversion
In [15]: df3['D'] = df3['D'].astype('float16')
```

(continues on next page)

(continued from previous page)

```
In [16]: df3['E'] = df3['E'].astype('int32')
```

```
In [17]: df3.dtypes
```

Out [17] :

A	float32
---	---------

B	float64
---	---------

C	float64
---	---------

D	float16
---	---------

E	int32
---	-------

```
Length: 5, dtype: object
```

### Forcing Date coercion (and setting NaT when not datelike)

```
In [18]: from datetime import datetime
```

```
In [19]: s = Series([datetime(2001,1,1,0,0), 'foo', 1.0, 1,
.....:               Timestamp('20010104'), '20010105'], dtype='O')
.....:
.....:
```

```
In [20]: s.convert_objects(convert_dates='coerce')
```

Out [20] :

0	2001-01-01
---	------------

1	NaT
---	-----

2	NaT
---	-----

3	NaT
---	-----

4	2001-01-04
---	------------

5	2001-01-05
---	------------

```
Length: 6, dtype: datetime64[ns]
```

### 1.28.5 Dtype Gotchas

## Platform Gotchas

Starting in 0.11.0, construction of DataFrame/Series will use default dtypes of `int64` and `float64`, *regardless of platform*. This is not an apparent change from earlier versions of pandas. If you specify dtypes, they *WILL* be respected, however ([GH2837](#))

The following will all result in `int64` dtypes

```
In [21]: DataFrame([1,2],columns=['a']).dtypes
```

Out [21] :

```
a      int64
```

```
Length: 1, dtype: object
```

```
In [22]: DataFrame({'a' : [1,2] }).dtypes
```

```
Out[22]:
```

```
a    int64
```

```
Length: 1, dtype: object
```

```
In [23]: DataFrame({'a' : 1 }, index=range(2)).dtypes
```



```
a      int64
```

```
Length: 1, dtype: object
```

Keep in mind that `DataFrame(np.array([1, 2]))` **WILL** result in `int32` on 32-bit platforms!

### Upcasting Gotchas

Performing indexing operations on integer type data can easily upcast the data. The dtype of the input data will be preserved in cases where nans are not introduced.

```
In [24]: dfi = df3.astype('int32')
```

```
In [25]: dfi['D'] = dfi['D'].astype('int64')
```

```
In [26]: dfi
```

```
Out[26]:
```

	A	B	C	D	E
0	1	0	0	1	1
1	0	0	1	1	1
2	0	0	0	1	1
3	-1	0	254	1	1
4	0	0	0	1	1
5	-1	0	1	1	1
6	2	1	1	1	1
7	-2	0	255	1	1

```
[8 rows x 5 columns]
```

```
In [27]: dfi.dtypes
```

```

////////////////////////////////////
↪
A      int32
B      int32
C      int32
D      int64
E      int32
Length: 5, dtype: object
```

```
In [28]: casted = dfi[dfi>0]
```

```
In [29]: casted
```

```
Out[29]:
```

	A	B	C	D	E
0	1.0	NaN	NaN	1	1
1	NaN	NaN	1.0	1	1
2	NaN	NaN	NaN	1	1
3	NaN	NaN	254.0	1	1
4	NaN	NaN	NaN	1	1
5	NaN	NaN	1.0	1	1
6	2.0	1.0	1.0	1	1
7	NaN	NaN	255.0	1	1

```
[8 rows x 5 columns]
```

```
In [30]: casted.dtypes
```

```

////////////////////////////////////
↪
A      float64
B      float64
C      float64
D      int64
E      int32
```

(continues on next page)



(continued from previous page)

```
Length: 5, dtype: object
```

While float dtypes are unchanged.

```
In [31]: df4 = df3.copy()

In [32]: df4['A'] = df4['A'].astype('float32')

In [33]: df4.dtypes
Out[33]:
A      float32
B      float64
C      float64
D      float16
E         int32
Length: 5, dtype: object
```

```
In [34]: casted = df4[df4>0]
```

```
In [35]: casted
Out[35]:
```

	A	B	C	D	E
0	1.984462	NaN	NaN	1.0	1
1	0.717812	NaN	1.0	1.0	1
2	NaN	NaN	NaN	1.0	1
3	NaN	0.866493	254.0	1.0	1
4	NaN	0.245972	NaN	1.0	1
5	NaN	0.319442	1.0	1.0	1
6	2.650092	1.378512	1.0	1.0	1
7	NaN	0.292502	255.0	1.0	1

```
[8 rows x 5 columns]
```

```
In [36]: casted.dtypes
```

```

////////////////////////////////////
↪
A      float32
B      float64
C      float64
D      float16
E         int32
Length: 5, dtype: object
```

## 1.28.6 Datetimes Conversion

Datetime64[ns] columns in a DataFrame (or a Series) allow the use of `np.nan` to indicate a nan value, in addition to the traditional NaT, or not-a-time. This allows convenient nan setting in a generic way. Furthermore datetime64[ns] columns are created by default, when passed datetimelike objects (*this change was introduced in 0.10.1*) (GH2809, GH2810)

```
In [37]: df = DataFrame(randn(6,2),date_range('20010102',periods=6),columns=['A','B'])

In [38]: df['timestamp'] = Timestamp('20010103')

In [39]: df
```

(continues on next page)

(continued from previous page)

Out [39]:

	A	B	timestamp
2001-01-02	1.023958	0.660103	2001-01-03
2001-01-03	1.236475	-2.170629	2001-01-03
2001-01-04	-0.270630	-1.685677	2001-01-03
2001-01-05	-0.440747	-0.115070	2001-01-03
2001-01-06	-0.632102	-0.585977	2001-01-03
2001-01-07	-1.444787	-0.201135	2001-01-03

[6 rows x 3 columns]

# datetime64[ns] out of the box

In [40]: df.get\_dtype\_counts()

```

////////////////////////////////////

```

↪

float64 2

datetime64[ns] 1

Length: 2, dtype: int64

# use the traditional nan, which is mapped to NaT internally

In [41]: df.loc[df.index[2:4], ['A', 'timestamp']] = np.nan

In [42]: df

Out [42]:

	A	B	timestamp
2001-01-02	1.023958	0.660103	2001-01-03
2001-01-03	1.236475	-2.170629	2001-01-03
2001-01-04	NaN	-1.685677	NaT
2001-01-05	NaN	-0.115070	NaT
2001-01-06	-0.632102	-0.585977	2001-01-03
2001-01-07	-1.444787	-0.201135	2001-01-03

[6 rows x 3 columns]

Astype conversion on datetime64[ns] to object, implicitly converts NaT to np.nan

In [43]: import datetime

In [44]: s = Series([datetime.datetime(2001, 1, 2, 0, 0) for i in range(3)])

In [45]: s.dtype

Out [45]: dtype('&lt;M8[ns]')

In [46]: s[1] = np.nan

In [47]: s

Out [47]:

0 2001-01-02

1 NaT

2 2001-01-02

Length: 3, dtype: datetime64[ns]

In [48]: s.dtype

```

//////////////////////////////////// Out [48]:

```

↪ dtype('&lt;M8[ns]')

In [49]: s = s.astype('O')

(continues on next page)

```
In [51]: s.dtypes
```

→ dtype('O')

```
In [55]: idx = date_range("2001-10-1", periods=5, freq='M')

In [56]: ts = Series(np.random.rand(len(idx)), index=idx)

In [57]: ts['2001']
Out[57]:
2001-10-31    0.663256
2001-11-30    0.079126
2001-12-31    0.587699
Freq: M, Length: 3, dtype: float64

In [58]: df = DataFrame(dict(A = ts))

In [59]: df['2001']
Out[59]:
           A
2001-10-31  0.663256
2001-11-30  0.079126
2001-12-31  0.587699

[3 rows x 1 columns]
```

- Squeeze to possibly remove length 1 dimensions from an object.

```
In [60]: p = Panel(randn(3,4,4),items=['ItemA','ItemB','ItemC'],
.....:               major_axis=date_range('20010102',periods=4),
.....:               minor_axis=['A','B','C','D'])
.....:

In [61]: p
Out[61]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D

In [62]: p.reindex(items=['ItemA']).squeeze()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
↩
      A      B      C      D
2001-01-02 -1.203403  0.425882 -0.436045 -0.982462
2001-01-03  0.348090 -0.969649  0.121731  0.202798
2001-01-04  1.215695 -0.218549 -0.631381 -0.337116
2001-01-05  0.404238  0.907213 -0.865657  0.483186

[4 rows x 4 columns]

In [63]: p.reindex(items=['ItemA'],minor=['B']).squeeze()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
↩
2001-01-02      0.425882
2001-01-03     -0.969649
2001-01-04     -0.218549
2001-01-05      0.907213
Freq: D, Name: B, Length: 4, dtype: float64
```

- In `pd.io.data.Options`,

- Fix bug when trying to fetch data for the current month when already past expiry.
- Now using lxml to scrape html instead of BeautifulSoup (lxml was faster).
- New instance variables for calls and puts are automatically created when a method that creates them is called. This works for current month where the instance variables are simply `calls` and `puts`. Also works for future expiry months and save the instance variable as `callsMMYY` or `putsMMYY`, where `MMYY` are, respectively, the month and year of the option’s expiry.
- `Options.get_near_stock_price` now allows the user to specify the month for which to get relevant options data.
- `Options.get_forward_data` now has optional kwargs `near` and `above_below`. This allows the user to specify if they would like to only return forward looking data for options near the current stock price. This just obtains the data from `Options.get_near_stock_price` instead of `Options.get_xxx_data()` (GH2758).
- Cursor coordinate information is now displayed in time-series plots.
- added option `display.max_seq_items` to control the number of elements printed per sequence pprinting it. (GH2979)
- added option `display.chop_threshold` to control display of small numerical values. (GH2739)
- added option `display.max_info_rows` to prevent verbose\_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- `value_counts()` now accepts a “normalize” argument, for normalized histograms. (GH2710).
- `DataFrame.from_records` now accepts not only dicts but any instance of the collections.Mapping ABC.
- added option `display.mpl_style` providing a sleeker visual style for plots. Based on <https://gist.github.com/huyng/816622> (GH3075).
- Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)
- `to_html()` now accepts an optional “escape” argument to control reserved HTML character escaping (enabled by default) and escapes `&`, in addition to `<` and `>`. (GH2919)

See the [full release notes](#) or issue tracker on GitHub for a complete list.

## 1.29 v0.10.1 (January 22, 2013)

This is a minor release from 0.10.0 and includes new features, enhancements, and bug fixes. In particular, there is substantial new HDFStore functionality contributed by Jeff Reback.

An undesired API breakage with functions taking the `inplace` option has been reverted and deprecation warnings added.

### 1.29.1 API changes

- Functions taking an `inplace` option return the calling object as before. A deprecation message has been added
- Groupby aggregations Max/Min no longer exclude non-numeric data (GH2700)
- Resampling an empty DataFrame now returns an empty DataFrame instead of raising an exception (GH2640)
- The file reader will now raise an exception when NA values are found in an explicitly specified integer column instead of converting the column to float (GH2631)
- `DatetimeIndex.unique` now returns a `DatetimeIndex` with the same name and

- timezone instead of an array ([GH2563](#))

### 1.29.2 New features

- MySQL support for database (contribution from Dan Allan)

### 1.29.3 HDFStore

You may need to upgrade your existing data files. Please visit the **compatibility** section in the main docs.

You can designate (and index) certain columns that you want to be able to perform queries on a table, by passing a list to `data_columns`

```
In [1]: store = HDFStore('store.h5')

In [2]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
....:                  columns=['A', 'B', 'C'])
....:

In [3]: df['string'] = 'foo'

In [4]: df.loc[df.index[4:6], 'string'] = np.nan

In [5]: df.loc[df.index[7:9], 'string'] = 'bar'

In [6]: df['string2'] = 'cool'

In [7]: df
Out[7]:
```

	A	B	C	string	string2
2000-01-01	1.885136	-0.183873	2.550850	foo	cool
2000-01-02	0.180759	-1.117089	0.061462	foo	cool
2000-01-03	-0.294467	-0.591411	-0.876691	foo	cool
2000-01-04	3.127110	1.451130	0.045152	foo	cool
2000-01-05	-0.242846	1.195819	1.533294	NaN	cool
2000-01-06	0.820521	-0.281201	1.651561	NaN	cool
2000-01-07	-0.034086	0.252394	-0.498772	foo	cool
2000-01-08	-2.290958	-1.601262	-0.256718	bar	cool

```
[8 rows x 5 columns]

# on-disk operations
In [8]: store.append('df', df, data_columns = ['B','C','string','string2'])

In [9]: store.select('df', "B>0 and string=='foo'")
Out[9]:
```

	A	B	C	string	string2
2000-01-04	3.127110	1.451130	0.045152	foo	cool
2000-01-07	-0.034086	0.252394	-0.498772	foo	cool

```
[2 rows x 5 columns]

# this is in-memory version of this type of selection
In [10]: df[(df.B > 0) & (df.string == 'foo')]
////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```

      A      B      C string string2
2000-01-04 3.127110 1.451130 0.045152    foo    cool
2000-01-07 -0.034086 0.252394 -0.498772    foo    cool

[2 rows x 5 columns]
```

Retrieving unique values in an indexable or data column.

```
# note that this is deprecated as of 0.14.0
# can be replicated by: store.select_column('df', 'index').unique()
store.unique('df', 'index')
store.unique('df', 'string')
```

You can now store `datetime64` in data columns

```
In [11]: df_mixed = df_mixed.copy()

In [12]: df_mixed['datetime64'] = Timestamp('20010102')

In [13]: df_mixed.loc[df_mixed.index[3:4], ['A', 'B']] = np.nan

In [14]: store.append('df_mixed', df_mixed)

In [15]: df_mixed1 = store.select('df_mixed')

In [16]: df_mixed1
Out[16]:
```

	A	B	C	string	string2	datetime64
2000-01-01	1.885136	-0.183873	2.550850	foo	cool	2001-01-02
2000-01-02	0.180759	-1.117089	0.061462	foo	cool	2001-01-02
2000-01-03	-0.294467	-0.591411	-0.876691	foo	cool	2001-01-02
2000-01-04	NaN	NaN	0.045152	foo	cool	2001-01-02
2000-01-05	-0.242846	1.195819	1.533294	NaN	cool	2001-01-02
2000-01-06	0.820521	-0.281201	1.651561	NaN	cool	2001-01-02
2000-01-07	-0.034086	0.252394	-0.498772	foo	cool	2001-01-02
2000-01-08	-2.290958	-1.601262	-0.256718	bar	cool	2001-01-02

```
[8 rows x 6 columns]

In [17]: df_mixed1.get_dtype_counts()
//////////
↪
float64      3
object       2
datetime64[ns] 1
Length: 3, dtype: int64
```

You can pass `columns` keyword to select to filter a list of the return columns, this is equivalent to passing a `Term('columns', list_of_columns_to_filter)`

```
In [18]: store.select('df', columns = ['A', 'B'])
Out[18]:
```

	A	B
2000-01-01	1.885136	-0.183873
2000-01-02	0.180759	-1.117089
2000-01-03	-0.294467	-0.591411

(continues on next page)

(continued from previous page)

```

2000-01-04  3.127110  1.451130
2000-01-05 -0.242846  1.195819
2000-01-06  0.820521 -0.281201
2000-01-07 -0.034086  0.252394
2000-01-08 -2.290958 -1.601262

```

```
[8 rows x 2 columns]
```

HDFStore now serializes multi-index dataframes when appending tables.

```

In [19]: index = MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
.....:                             ['one', 'two', 'three']],
.....:                       labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
.....:                              [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
.....:                       names=['foo', 'bar'])
.....:

```

```

In [20]: df = DataFrame(np.random.randn(10, 3), index=index,
.....:                  columns=['A', 'B', 'C'])
.....:

```

```
In [21]: df
```

```

Out[21]:
           A           B           C
foo bar
foo one    0.239369    0.174122 -1.131794
   two   -1.948006    0.980347 -0.674429
   three -0.361633   -0.761218  1.768215
bar one    0.152288   -0.862613 -0.210968
   two   -0.859278    1.498195  0.462413
baz two   -0.647604    1.511487 -0.727189
   three -0.342928   -0.007364  1.427674
qux one    0.104020    2.052171 -1.230963
   two   -0.019240   -1.713238  0.838912
   three -0.637855    0.215109 -1.515362

```

```
[10 rows x 3 columns]
```

```
In [22]: store.append('mi', df)
```

```
In [23]: store.select('mi')
```

```

Out[23]:
           A           B           C
foo bar
foo one    0.239369    0.174122 -1.131794
   two   -1.948006    0.980347 -0.674429
   three -0.361633   -0.761218  1.768215
bar one    0.152288   -0.862613 -0.210968
   two   -0.859278    1.498195  0.462413
baz two   -0.647604    1.511487 -0.727189
   three -0.342928   -0.007364  1.427674
qux one    0.104020    2.052171 -1.230963
   two   -0.019240   -1.713238  0.838912
   three -0.637855    0.215109 -1.515362

```

```
[10 rows x 3 columns]
```

(continues on next page)



(continued from previous page)

# the levels are automatically included as data columns

**In [24]:** store.select('mi', "foo='bar'")

```

////////////////////////////////////
↪
           A          B          C
foo bar
bar one  0.152288 -0.862613 -0.210968
      two -0.859278  1.498195  0.462413

```

[2 rows x 3 columns]

Multi-table creation via `append_to_multiple` and selection via `select_as_multiple` can create/select from multiple tables and return a combined result, by using `where` on a selector table.

```

In [25]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
.....:                  columns=['A', 'B', 'C', 'D', 'E', 'F'])
.....:

```

**In [26]:** df\_mt['foo'] = 'bar'

# you can also create the tables individually

```

In [27]: store.append_to_multiple({'df1_mt' : ['A', 'B'], 'df2_mt' : None }, df_mt,
↪selector = 'df1_mt')

```

**In [28]:** store**Out [28]:**

&lt;class 'pandas.io.pytables.HDFStore'&gt;

File path: store.h5

# individual tables were created

**In [29]:** store.select('df1\_mt')

```

////////////////////////////////////Out [29]:

```

```

           A          B
2000-01-01  1.586924 -0.447974
2000-01-02 -0.102206  0.870302
2000-01-03  1.249874  1.458210
2000-01-04 -0.616293  0.150468
2000-01-05 -0.431163  0.016640
2000-01-06  0.800353 -0.451572
2000-01-07  1.239198  0.185437
2000-01-08 -0.040863  0.290110

```

[8 rows x 2 columns]

**In [30]:** store.select('df2\_mt')

```

////////////////////////////////////
↪
           C          D          E          F  foo
2000-01-01 -1.573998  0.630925 -0.071659 -1.277640  bar
2000-01-02  1.275280 -1.199212  1.060780  1.673018  bar
2000-01-03 -0.710542  0.825392  1.557329  1.993441  bar
2000-01-04  0.132104  0.580923 -0.128750  1.445964  bar
2000-01-05  0.904578 -1.645852 -0.688741  0.228006  bar
2000-01-06  0.831767  0.228760  0.932498 -2.200069  bar
2000-01-07 -0.540770 -0.370038  1.298390  1.662964  bar
2000-01-08 -0.096145  1.717830 -0.462446 -0.112019  bar

```

(continues on next page)

(continued from previous page)

```
[8 rows x 5 columns]

# as a multiple
In [31]: store.select_as_multiple(['df1_mt', 'df2_mt'], where = [ 'A>0', 'B>0' ],
↳ selector = 'df1_mt')
////////////////////////////////////
↳
           A          B          C          D          E          F    foo
2000-01-03  1.249874  1.458210 -0.710542  0.825392  1.557329  1.993441  bar
2000-01-07  1.239198  0.185437 -0.540770 -0.370038  1.298390  1.662964  bar

[2 rows x 7 columns]
```

## Enhancements

- `HDFStore` now can read native PyTables table format tables
- You can pass `nan_rep = 'my_nan_rep'` to `append`, to change the default nan representation on disk (which converts to/from `np.nan`), this defaults to `nan`.
- You can pass `index` to `append`. This defaults to `True`. This will automatically create indices on the *indexables* and *data columns* of the table
- You can pass `chunksize=an integer` to `append`, to change the writing chunksize (default is 50000). This will significantly lower your memory usage on writing.
- You can pass `expectedrows=an integer` to the first `append`, to set the TOTAL number of expected rows that PyTables will expect. This will optimize read/write performance.
- `Select` now supports passing `start` and `stop` to provide selection space limiting in selection.
- Greatly improved ISO8601 (e.g., yyyy-mm-dd) date parsing for file parsers ([GH2698](#))
- Allow `DataFrame.merge` to handle combinatorial sizes too large for 64-bit integer ([GH2690](#))
- `Series` now has unary negation (`-series`) and inversion (`~series`) operators ([GH2686](#))
- `DataFrame.plot` now includes a `logx` parameter to change the x-axis to log scale ([GH2327](#))
- `Series` arithmetic operators can now handle constant and `ndarray` input ([GH2574](#))
- `ExcelFile` now takes a `kind` argument to specify the file type ([GH2613](#))
- A faster implementation for `Series.str` methods ([GH2602](#))

## Bug Fixes

- `HDFStore` tables can now store `float32` types correctly (cannot be mixed with `float64` however)
- Fixed Google Analytics prefix when specifying request segment ([GH2713](#)).
- Function to reset Google Analytics token store so users can recover from improperly setup client secrets ([GH2687](#)).
- Fixed `groupby` bug resulting in segfault when passing in `MultiIndex` ([GH2706](#))
- Fixed bug where passing a `Series` with `datetime64` values into `to_datetime` results in bogus output values ([GH2699](#))
- Fixed bug in `pattern` in `HDFStore` expressions when `pattern` is not a valid regex ([GH2694](#))
- Fixed performance issues while aggregating boolean data ([GH2692](#))

- When given a boolean mask key and a Series of new values, Series `__setitem__` will now align the incoming values with the original Series ([GH2686](#))
- Fixed MemoryError caused by performing counting sort on sorting MultiIndex levels with a very large number of combinatorial values ([GH2684](#))
- Fixed bug that causes plotting to fail when the index is a DatetimeIndex with a fixed-offset timezone ([GH2683](#))
- Corrected businessday subtraction logic when the offset is more than 5 bdays and the starting date is on a weekend ([GH2680](#))
- Fixed C file parser behavior when the file has more columns than data ([GH2668](#))
- Fixed file reader bug that misaligned columns with data in the presence of an implicit column and a specified *usecols* value
- DataFrames with numerical or datetime indices are now sorted prior to plotting ([GH2609](#))
- Fixed DataFrame.from\_records error when passed columns, index, but empty records ([GH2633](#))
- Several bug fixed for Series operations when dtype is datetime64 ([GH2689](#), [GH2629](#), [GH2626](#))

See the [full release notes](#) or issue tracker on GitHub for a complete list.

## 1.30 v0.10.0 (December 17, 2012)

This is a major release from 0.9.1 and includes many new features and enhancements along with a large number of bug fixes. There are also a number of important API changes that long-time pandas users should pay close attention to.

### 1.30.1 File parsing new features

The delimited file parsing engine (the guts of `read_csv` and `read_table`) has been rewritten from the ground up and now uses a fraction the amount of memory while parsing, while being 40% or more faster in most use cases (in some cases much faster).

There are also many new features:

- Much-improved Unicode handling via the `encoding` option.
- Column filtering (`usecols`)
- Dtype specification (`dtype` argument)
- Ability to specify strings to be recognized as True/False
- Ability to yield NumPy record arrays (`as_recarray`)
- High performance `delim_whitespace` option
- Decimal format (e.g. European format) specification
- Easier CSV dialect options: `escapechar`, `lineterminator`, `quotechar`, etc.
- More robust handling of many exceptional kinds of files observed in the wild

### 1.30.2 API changes

## Deprecated DataFrame BINOP TimeSeries special case behavior

The default behavior of binary operations between a DataFrame and a Series has always been to align on the DataFrame's columns and broadcast down the rows, **except** in the special case that the DataFrame contains time series. Since there are now method for each binary operator enabling you to specify how you want to broadcast, we are phasing out this special case (*Zen of Python: Special cases aren't special enough to break the rules*). Here's what I'm talking about:

```
In [1]: import pandas as pd

In [2]: df = pd.DataFrame(np.random.randn(6, 4),
...:                      index=pd.date_range('1/1/2000', periods=6))
...:
```

```
In [3]: df
```

```
Out[3]:
```

	0	1	2	3
2000-01-01	-0.134024	-0.205969	1.348944	-1.198246
2000-01-02	-1.626124	0.982041	0.059493	-0.460111
2000-01-03	-1.565401	-0.025706	0.942864	2.502156
2000-01-04	-0.302741	0.261551	-0.066342	0.897097
2000-01-05	0.268766	-1.225092	0.582752	-1.490764
2000-01-06	-0.639757	-0.952750	-0.892402	0.505987

```
[6 rows x 4 columns]
```

```
# deprecated now
In [4]: df - df[0]
~~~~~
↪
      2000-01-01 00:00:00    2000-01-02 00:00:00    2000-01-03 00:00:00
↪00:00:00 ...      0      1      2      3
2000-01-01          NaN                                NaN
↪      NaN ... NaN NaN NaN NaN
2000-01-02          NaN                                NaN
↪      NaN ... NaN NaN NaN NaN
2000-01-03          NaN                                NaN
↪      NaN ... NaN NaN NaN NaN
2000-01-04          NaN                                NaN
↪      NaN ... NaN NaN NaN NaN
2000-01-05          NaN                                NaN
↪      NaN ... NaN NaN NaN NaN
2000-01-06          NaN                                NaN
↪      NaN ... NaN NaN NaN NaN

[6 rows x 10 columns]
```

```
# Change your code to
In [5]: df.sub(df[0], axis=0) # align on axis 0 (rows)
~~~~~
↪
      0      1      2      3
2000-01-01  0.0 -0.071946  1.482967 -1.064223
2000-01-02  0.0  2.608165  1.685618  1.166013
2000-01-03  0.0  1.539695  2.508265  4.067556
2000-01-04  0.0  0.564293  0.236399  1.199839
```

(continues on next page)

(continued from previous page)

```

2000-01-05    0.0 -1.493857    0.313986 -1.759530
2000-01-06    0.0 -0.312993   -0.252645    1.145744

[6 rows x 4 columns]

```

You will get a deprecation warning in the 0.10.x series, and the deprecated functionality will be removed in 0.11 or later.

### Altered resample default behavior

The default time series `resample` binning behavior of daily `D` and *higher* frequencies has been changed to `closed='left', label='left'`. Lower frequencies are unaffected. The prior defaults were causing a great deal of confusion for users, especially resampling data to daily frequency (which labeled the aggregated group with the end of the interval: the next day).

```
In [1]: dates = pd.date_range('1/1/2000', '1/5/2000', freq='4h')
```

```
In [2]: series = Series(np.arange(len(dates)), index=dates)
```

```
In [3]: series
```

```
Out [3]:
```

```

2000-01-01 00:00:00      0
2000-01-01 04:00:00      1
2000-01-01 08:00:00      2
2000-01-01 12:00:00      3
2000-01-01 16:00:00      4
2000-01-01 20:00:00      5
2000-01-02 00:00:00      6
2000-01-02 04:00:00      7
2000-01-02 08:00:00      8
2000-01-02 12:00:00      9
2000-01-02 16:00:00     10
2000-01-02 20:00:00     11
2000-01-03 00:00:00     12
2000-01-03 04:00:00     13
2000-01-03 08:00:00     14
2000-01-03 12:00:00     15
2000-01-03 16:00:00     16
2000-01-03 20:00:00     17
2000-01-04 00:00:00     18
2000-01-04 04:00:00     19
2000-01-04 08:00:00     20
2000-01-04 12:00:00     21
2000-01-04 16:00:00     22
2000-01-04 20:00:00     23
2000-01-05 00:00:00     24
Freq: 4H, dtype: int64

```

```
In [4]: series.resample('D', how='sum')
```

```
Out [4]:
```

```

2000-01-01      15
2000-01-02      51
2000-01-03      87
2000-01-04     123
2000-01-05      24
Freq: D, dtype: int64

```

(continues on next page)

(continued from previous page)

```

In [5]: # old behavior
In [6]: series.resample('D', how='sum', closed='right', label='right')
Out[6]:
2000-01-01      0
2000-01-02     21
2000-01-03     57
2000-01-04     93
2000-01-05    129
Freq: D, dtype: int64

```

- Infinity and negative infinity are no longer treated as NA by `isnull` and `notnull`. That they ever were was a relic of early pandas. This behavior can be re-enabled globally by the `mode.use_inf_as_null` option:

```

In [6]: s = pd.Series([1.5, np.inf, 3.4, -np.inf])

In [7]: pd.isnull(s)
Out[7]:
0    False
1    False
2    False
3    False
Length: 4, dtype: bool

In [8]: s.fillna(0)
Out[8]:
0    1.500000
1         inf
2    3.400000
3        -inf
Length: 4, dtype: float64

In [9]: pd.set_option('use_inf_as_null', True)

In [10]: pd.isnull(s)
Out[10]:
0    False
1     True
2    False
3     True
Length: 4, dtype: bool

In [11]: s.fillna(0)
Out[11]:
0    1.5
1    0.0
2    3.4
3    0.0
Length: 4, dtype: float64

In [12]: pd.reset_option('use_inf_as_null')

```

- Methods with the `inplace` option now all return `None` instead of the calling object. E.g. code written like `df = df.fillna(0, inplace=True)` may stop working. To fix, simply delete the unnecessary variable assignment.
- `pandas.merge` no longer sorts the group keys (`sort=False`) by default. This was done for performance reasons: the group-key sorting is often one of the more expensive parts of the computation and is often unne-

essary.

- The default column names for a file with no header have been changed to the integers 0 through  $N - 1$ . This is to create consistency with the DataFrame constructor with no columns specified. The v0.9.0 behavior (names X0, X1, ...) can be reproduced by specifying `prefix='X'`:

```
In [6]: data= 'a,b,c\n1,Yes,2\n3,No,4'

In [7]: print(data)
a,b,c
1,Yes,2
3,No,4

In [8]: pd.read_csv(StringIO(data), header=None)
\\Out[8]:
   0  1  2
0  a  b  c
1  1  Yes 2
2  3  No  4

[3 rows x 3 columns]

In [9]: pd.read_csv(StringIO(data), header=None, prefix='X')
\\Out[9]:
↪
   X0  X1 X2
0  a   b  c
1  1  Yes 2
2  3  No  4

[3 rows x 3 columns]
```

- Values like 'Yes' and 'No' are not interpreted as boolean by default, though this can be controlled by new `true_values` and `false_values` arguments:

```
In [10]: print(data)
a,b,c
1,Yes,2
3,No,4

In [11]: pd.read_csv(StringIO(data))
\\Out[11]:
   a  b  c
0  1  Yes 2
1  3  No  4

[2 rows x 3 columns]

In [12]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
\\Out[12]:
↪
   a  b  c
0  1  True 2
1  3  False 4

[2 rows x 3 columns]
```

- The file parsers will not recognize non-string values arising from a converter function as NA if passed in the `na_values` argument. It's better to do post-processing using the `replace` function instead.

- Calling `fillna` on `Series` or `DataFrame` with no arguments is no longer valid code. You must either specify a fill value or an interpolation method:

```
In [13]: s = Series([np.nan, 1., 2., np.nan, 4])

In [14]: s
Out[14]:
0      NaN
1      1.0
2      2.0
3      NaN
4      4.0
Length: 5, dtype: float64

In [15]: s.fillna(0)
Out[15]:
0      0.0
1      1.0
2      2.0
3      0.0
4      4.0
Length: 5, dtype: float64

In [16]: s.fillna(method='pad')
Out[16]:
0      NaN
1      1.0
2      2.0
3      2.0
4      4.0
Length: 5, dtype: float64
```

Convenience methods `ffill` and `bfill` have been added:

```
In [17]: s.fffll()
Out[17]:
0      NaN
1      1.0
2      2.0
3      2.0
4      4.0
Length: 5, dtype: float64
```

- `Series.apply` will now operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a `DataFrame`

```
In [18]: def f(x):
.....:     return Series([ x, x**2 ], index = ['x', 'x^2'])
.....:

In [19]: s = Series(np.random.rand(5))

In [20]: s
Out[20]:
0      0.717478
1      0.815199
```

(continues on next page)



(continued from previous page)

```

2    0.452478
3    0.848385
4    0.235477
Length: 5, dtype: float64

```

```
In [21]: s.apply(f)
```

```

////////////////////////////////////
↪
      x      x^2
0  0.717478  0.514775
1  0.815199  0.664550
2  0.452478  0.204737
3  0.848385  0.719757
4  0.235477  0.055449

[5 rows x 2 columns]

```

- New API functions for working with pandas options ([GH2097](#)):

- `get_option` / `set_option` - get/set the value of an option. Partial names are accepted.
- `reset_option` - reset one or more options to their default value. Partial names are accepted.
- `describe_option` - print a description of one or more options. When called with no arguments, print all registered options.

Note: `set_printoptions` / `reset_printoptions` are now deprecated (but functioning), the print options now live under “`display.XYZ`”. For example:

```
In [22]: get_option("display.max_rows")
Out [22]: 15
```

- `to_string()` methods now always return unicode strings ([GH2224](#)).

### 1.30.3 New features

#### 1.30.4 Wide DataFrame Printing

Instead of printing the summary information, pandas now splits the string representation across multiple rows by default:

```
In [23]: wide_frame = DataFrame(randn(5, 16))
```

```
In [24]: wide_frame
```

```

Out [24]:
      0         1         2         3         4         ...         11         12
↪  13         14         15
0 -0.681624  0.191356  1.180274 -0.834179  0.703043  ... -0.941235  0.863067 -0.
↪ 336232 -0.976847  0.033862
1  0.441522 -0.316864 -0.017062  1.570114 -0.360875  ... -0.906840  1.014601 -0.
↪ 475108 -0.358944  1.262942
2 -0.412451 -0.462580  0.422194  0.288403 -0.487393  ...  0.246354 -0.727728 -0.
↪ 094414 -0.276854  0.158399
3 -0.277255  1.331263  0.585174 -0.568825 -0.719412  ... -1.425911 -0.548829  0.
↪ 774225  0.740501  1.510263
4 -1.642511  0.432560  1.218080 -0.564705 -0.581790  ... -0.471786  0.314510 -0.
↪ 059986 -2.069319 -1.115104

```

(continues on next page)

(continued from previous page)

[5 rows x 16 columns]

The old behavior of printing out summary information can be achieved via the ‘expand\_frame\_repr’ print option:

```
In [25]: pd.set_option('expand_frame_repr', False)
```

```
In [26]: wide_frame
```

```
Out[26]:
```

```

      0         1         2         3         4         5         6         7
→  8      9      10      11      12      13      14      15
0 -0.681624  0.191356  1.180274 -0.834179  0.703043  0.166568 -0.583599 -1.201796 -1.
→ 422811 -0.882554  1.209871 -0.941235  0.863067 -0.336232 -0.976847  0.033862
1  0.441522 -0.316864 -0.017062  1.570114 -0.360875 -0.880096  0.235532  0.207232 -1.
→ 983857 -1.702547 -1.621234 -0.906840  1.014601 -0.475108 -0.358944  1.262942
2 -0.412451 -0.462580  0.422194  0.288403 -0.487393 -0.777639  0.055865  1.383381  0.
→ 085638  0.246392  0.965887  0.246354 -0.727728 -0.094414 -0.276854  0.158399
3 -0.277255  1.331263  0.585174 -0.568825 -0.719412  1.191340 -0.456362  0.089931  0.
→ 776079  0.752889 -1.195795 -1.425911 -0.548829  0.774225  0.740501  1.510263
4 -1.642511  0.432560  1.218080 -0.564705 -0.581790  0.286071  0.048725  1.002440  1.
→ 276582  0.054399  0.241963 -0.471786  0.314510 -0.059986 -2.069319 -1.115104

[5 rows x 16 columns]
```

The width of each line can be changed via ‘line\_width’ (80 by default):

```
In [27]: pd.set_option('line_width', 40)
```

```

-----
OptionError                                Traceback (most recent call last)
<ipython-input-27-b8740c4a0a1b> in <module>()
----> 1 pd.set_option('line_width', 40)

~/sandbox/pandas-release/pandas-docs/pandas/core/config.py in __call__(self, *args, _
→ **kwargs)
    225
    226     def __call__(self, *args, **kwargs):
--> 227         return self.__func__(*args, **kwargs)
    228
    229     @property

~/sandbox/pandas-release/pandas-docs/pandas/core/config.py in _set_option(*args, _
→ **kwargs)
    117
    118     for k, v in zip(args[::2], args[1::2]):
--> 119         key = _get_single_key(k, silent)
    120
    121         o = _get_registered_option(key)

~/sandbox/pandas-release/pandas-docs/pandas/core/config.py in _get_single_key(pat, _
→ silent)
    81         if not silent:
    82             _warn_if_deprecated(pat)
--> 83         raise OptionError('No such keys(s): {pat!r}'.format(pat=pat))
    84     if len(keys) > 1:
    85         raise OptionError('Pattern matched multiple keys')

OptionError: "No such keys(s): 'line_width'"

```

(continues on next page)

(continued from previous page)

**In [28]:** wide\_frame

```

////////////////////////////////////
↪
      0      1      2      3      4      ...      11      12
↪  13      14      15
0 -0.681624  0.191356  1.180274 -0.834179  0.703043  ... -0.941235  0.863067 -0.
↪ 336232 -0.976847  0.033862
1  0.441522 -0.316864 -0.017062  1.570114 -0.360875  ... -0.906840  1.014601 -0.
↪ 475108 -0.358944  1.262942
2 -0.412451 -0.462580  0.422194  0.288403 -0.487393  ...  0.246354 -0.727728 -0.
↪ 094414 -0.276854  0.158399
3 -0.277255  1.331263  0.585174 -0.568825 -0.719412  ... -1.425911 -0.548829  0.
↪ 774225  0.740501  1.510263
4 -1.642511  0.432560  1.218080 -0.564705 -0.581790  ... -0.471786  0.314510 -0.
↪ 059986 -2.069319 -1.115104

[5 rows x 16 columns]

```

### 1.30.5 Updated PyTables Support

*Docs* for PyTables Table format & several enhancements to the api. Here is a taste of what to expect.

**In [29]:** store = HDFStore('store.h5')

```

In [30]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
.....:                  columns=['A', 'B', 'C'])
.....:

```

**In [31]:** df**Out[31]:**

```

      A      B      C
2000-01-01 -0.369325 -1.502617 -0.376280
2000-01-02  0.511936 -0.116412 -0.625256
2000-01-03 -0.550627  1.261433 -0.552429
2000-01-04  1.695803 -1.025917 -0.910942
2000-01-05  0.426805 -0.131749  0.432600
2000-01-06  0.044671 -0.341265  1.844536
2000-01-07 -2.036047  0.000830 -0.955697
2000-01-08 -0.898872 -0.725411  0.059904

```

[8 rows x 3 columns]

# appending data frames

**In [32]:** df1 = df[0:4]**In [33]:** df2 = df[4:]**In [34]:** store.append('df', df1)**In [35]:** store.append('df', df2)**In [36]:** store**Out[36]:**

&lt;class 'pandas.io.pytables.HDFStore'&gt;

(continues on next page)

(continued from previous page)

File path: store.h5

# selecting the entire store

**In [37]:** store.select('df')

Out[37]:

	A	B	C
2000-01-01	-0.369325	-1.502617	-0.376280
2000-01-02	0.511936	-0.116412	-0.625256
2000-01-03	-0.550627	1.261433	-0.552429
2000-01-04	1.695803	-1.025917	-0.910942
2000-01-05	0.426805	-0.131749	0.432600
2000-01-06	0.044671	-0.341265	1.844536
2000-01-07	-2.036047	0.000830	-0.955697
2000-01-08	-0.898872	-0.725411	0.059904

[8 rows x 3 columns]

**In [38]:** wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],

.....: major\_axis=date\_range('1/1/2000', periods=5),

.....: minor\_axis=['A', 'B', 'C', 'D'])

.....:

**In [39]:** wp**Out[39]:**

&lt;class 'pandas.core.panel.Panel'&gt;

Dimensions: 2 (items) x 5 (major\_axis) x 4 (minor\_axis)

Items axis: Item1 to Item2

Major\_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00

Minor\_axis axis: A to D

# storing a panel

**In [40]:** store.append('wp', wp)

# selecting via A QUERY

**In [41]:** store.select('wp', "major\_axis>20000102 and minor\_axis=['A','B']")**Out[41]:**

&lt;class 'pandas.core.panel.Panel'&gt;

Dimensions: 2 (items) x 3 (major\_axis) x 2 (minor\_axis)

Items axis: Item1 to Item2

Major\_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00

Minor\_axis axis: A to B

# removing data from tables

**In [42]:** store.remove('wp', "major\_axis>20000103")

Out[42]:

↪ 8

**In [43]:** store.select('wp')

Out[43]:

↪

&lt;class 'pandas.core.panel.Panel'&gt;

Dimensions: 2 (items) x 3 (major\_axis) x 4 (minor\_axis)

Items axis: Item1 to Item2

Major\_axis axis: 2000-01-01 00:00:00 to 2000-01-03 00:00:00

Minor\_axis axis: A to D

(continues on next page)

(continued from previous page)

```
# deleting a store
In [44]: del store['df']

In [45]: store
Out[45]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

### Enhancements

- added ability to hierarchical keys

```
In [46]: store.put('foo/bar/bah', df)

In [47]: store.append('food/orange', df)

In [48]: store.append('food/apple', df)

In [49]: store
Out[49]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# remove all nodes under this level
In [50]: store.remove('food')

In [51]: store
Out[51]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

- added mixed-dtype support!

```
In [52]: df['string'] = 'string'

In [53]: df['int'] = 1

In [54]: store.append('df', df)

In [55]: df1 = store.select('df')

In [56]: df1
Out[56]:
```

	A	B	C	string	int
2000-01-01	-0.369325	-1.502617	-0.376280	string	1
2000-01-02	0.511936	-0.116412	-0.625256	string	1
2000-01-03	-0.550627	1.261433	-0.552429	string	1
2000-01-04	1.695803	-1.025917	-0.910942	string	1
2000-01-05	0.426805	-0.131749	0.432600	string	1
2000-01-06	0.044671	-0.341265	1.844536	string	1
2000-01-07	-2.036047	0.000830	-0.955697	string	1
2000-01-08	-0.898872	-0.725411	0.059904	string	1

```
[8 rows x 5 columns]

In [57]: df1.get_dtype_counts()
//////////
```

(continues on next page)

(continued from previous page)

```
float64    3
object     1
int64      1
Length: 3, dtype: int64
```

- performance improvements on table writing
- support for arbitrarily indexed dimensions
- `SparseSeries` now has a density property ([GH2384](#))
- enable `Series.str.strip/lstrip/rstrip` methods to take an input argument to strip arbitrary characters ([GH2411](#))
- implement `value_vars` in `melt` to limit values to certain columns and add `melt` to pandas namespace ([GH2412](#))

### Bug Fixes

- added `Term` method of specifying where conditions ([GH1996](#)).
- `del store['df']` now call `store.remove('df')` for store deletion
- deleting of consecutive rows is much faster than before
- `min_itemsize` parameter can be specified in table creation to force a minimum size for indexing columns (the previous implementation would set the column size based on the first append)
- indexing support via `create_table_index` (requires `PyTables >= 2.3`) ([GH698](#)).
- appending on a store would fail if the table was not first created via `put`
- fixed issue with missing attributes after loading a pickled dataframe ([GH2431](#))
- minor change to `select` and `remove`: require a table ONLY if where is also provided (and not `None`)

### Compatibility

0.10 of `HDFStore` is backwards compatible for reading tables created in a prior version of pandas, however, query terms using the prior (undocumented) methodology are unsupported. You must read in the entire file and write it out using the new format to take advantage of the updates.

## 1.30.6 N Dimensional Panels (Experimental)

Adding experimental support for `Panel4D` and factory functions to create n-dimensional named panels. Here is a taste of what to expect.

```
In [58]: p4d = Panel4D(randn(2, 2, 5, 4),
....:      labels=['Label1', 'Label2'],
....:      items=['Item1', 'Item2'],
....:      major_axis=date_range('1/1/2000', periods=5),
....:      minor_axis=['A', 'B', 'C', 'D'])
....:

In [59]: p4d
Out[59]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
Items axis: Item1 to Item2
```

(continues on next page)

(continued from previous page)

```
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

See the *full release notes* or issue tracker on GitHub for a complete list.

## 1.31 v0.9.1 (November 14, 2012)

This is a bugfix release from 0.9.0 and includes several new features and enhancements along with a large number of bug fixes. The new features include by-column sort order for DataFrame and Series, improved NA handling for the rank method, masking functions for DataFrame, and intraday time-series filtering for DataFrame.

### 1.31.1 New features

- `Series.sort`, `DataFrame.sort`, and `DataFrame.sort_index` can now be specified in a per-column manner to support multiple sort orders ([GH928](#))

```
In [2]: df = DataFrame(np.random.randint(0, 2, (6, 3)), columns=['A', 'B', 'C'])
In [3]: df.sort(['A', 'B'], ascending=[1, 0])

Out[3]:
```

	A	B	C
3	0	1	1
4	0	1	1
2	0	0	1
0	1	0	0
1	1	0	0
5	1	0	0

- `DataFrame.rank` now supports additional argument values for the `na_option` parameter so missing values can be assigned either the largest or the smallest rank ([GH1508](#), [GH2159](#))

```
In [1]: df = DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])

In [2]: df.loc[2:4] = np.nan

In [3]: df.rank()
Out[3]:
```

	A	B	C
0	3.0	1.0	3.0
1	1.0	3.0	2.0
2	NaN	NaN	NaN
3	NaN	NaN	NaN
4	NaN	NaN	NaN
5	2.0	2.0	1.0

```
[6 rows x 3 columns]

In [4]: df.rank(na_option='top')
```

```
////////////////////////////////////
```

```
↪
```

	A	B	C
0	6.0	4.0	6.0

(continues on next page)

(continued from previous page)

```

1  4.0  6.0  5.0
2  2.0  2.0  2.0
3  2.0  2.0  2.0
4  2.0  2.0  2.0
5  5.0  5.0  4.0

```

```
[6 rows x 3 columns]
```

```
In [5]: df.rank(na_option='bottom')
```

```

////////////////////////////////////
↪
      A      B      C
0  3.0  1.0  3.0
1  1.0  3.0  2.0
2  5.0  5.0  5.0
3  5.0  5.0  5.0
4  5.0  5.0  5.0
5  2.0  2.0  1.0

```

```
[6 rows x 3 columns]
```

- DataFrame has new *where* and *mask* methods to select values according to a given boolean mask ([GH2109](#), [GH2151](#))

DataFrame currently supports slicing via a boolean vector the same length as the DataFrame (inside the `[]`). The returned DataFrame has the same number of columns as the original, but is sliced on its index.

```
In [6]: df = DataFrame(np.random.randn(5, 3), columns = ['A', 'B', 'C'])
```

```
In [7]: df
```

```
Out[7]:
      A      B      C
0 -1.101581 -1.187831  0.630693
1  2.369983  0.333769 -0.870464
2  1.118760 -0.224382  0.642489
3  0.961751 -1.848369  0.440883
4  1.235390  1.615529 -0.303272

```

```
[5 rows x 3 columns]
```

```
In [8]: df[df['A'] > 0]
```

```

////////////////////////////////////
↪
      A      B      C
1  2.369983  0.333769 -0.870464
2  1.118760 -0.224382  0.642489
3  0.961751 -1.848369  0.440883
4  1.235390  1.615529 -0.303272

```

```
[4 rows x 3 columns]
```

If a DataFrame is sliced with a DataFrame based boolean condition (with the same size as the original DataFrame), then a DataFrame the same size (index and columns) as the original is returned, with elements that do not meet the boolean condition as *NaN*. This is accomplished via the new method *DataFrame.where*. In addition, *where* takes an optional *other* argument for replacement.



```
In [9]: df[df>0]
```

```
Out[9]:
```

	A	B	C
0	NaN	NaN	0.630693
1	2.369983	0.333769	NaN
2	1.118760	NaN	0.642489
3	0.961751	NaN	0.440883
4	1.235390	1.615529	NaN

```
[5 rows x 3 columns]
```

```
In [10]: df.where(df>0)
```

```

////////////////////////////////////
↪
      A      B      C
0    NaN    NaN  0.630693
1  2.369983  0.333769    NaN
2  1.118760    NaN  0.642489
3  0.961751    NaN  0.440883
4  1.235390  1.615529    NaN

```

```
[5 rows x 3 columns]
```

```
In [11]: df.where(df>0,-df)
```

```

////////////////////////////////////
↪
      A      B      C
0  1.101581  1.187831  0.630693
1  2.369983  0.333769  0.870464
2  1.118760  0.224382  0.642489
3  0.961751  1.848369  0.440883
4  1.235390  1.615529  0.303272

```

```
[5 rows x 3 columns]
```

Furthermore, *where* now aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via *.ix* (but on the contents rather than the axis labels)

```
In [12]: df2 = df.copy()
```

```
In [13]: df2[ df2[1:4] > 0 ] = 3
```

```
In [14]: df2
```

```
Out[14]:
```

	A	B	C
0	-1.101581	-1.187831	0.630693
1	3.000000	3.000000	-0.870464
2	3.000000	-0.224382	3.000000
3	3.000000	-1.848369	3.000000
4	1.235390	1.615529	-0.303272

```
[5 rows x 3 columns]
```

*DataFrame.mask* is the inverse boolean operation of *where*.

```
In [15]: df.mask(df<=0)
```

(continues on next page)

(continued from previous page)

```
Out [15]:
```

	A	B	C
0	NaN	NaN	0.630693
1	2.369983	0.333769	NaN
2	1.118760	NaN	0.642489
3	0.961751	NaN	0.440883
4	1.235390	1.615529	NaN

```
[5 rows x 3 columns]
```

- Enable referencing of Excel columns by their column names ([GH1936](#))

```
In [16]: xl = ExcelFile('data/test.xls')

In [17]: xl.parse('Sheet1', index_col=0, parse_dates=True,
.....:           parse_cols='A:D')
.....:
Out [17]:
```

	A	B	C	D
2000-01-03	0.980269	3.685731	-0.364217	-1.159738
2000-01-04	1.047916	-0.041232	-0.161812	0.212549
2000-01-05	0.498581	0.731168	-0.537677	1.346270
2000-01-06	1.120202	1.567621	0.003641	0.675253
2000-01-07	-0.487094	0.571455	-1.611639	0.103469
2000-01-10	0.836649	0.246462	0.588543	1.062782
2000-01-11	-0.157161	1.340307	1.195778	-1.097007

```
[7 rows x 4 columns]
```

- Added option to disable pandas-style tick locators and formatters using `series.plot(x_compat=True)` or `pandas.plot_params['x_compat'] = True` ([GH2205](#))
- Existing TimeSeries methods `at_time` and `between_time` were added to DataFrame ([GH2149](#))
- DataFrame.dot can now accept ndarrays ([GH2042](#))
- DataFrame.drop now supports non-unique indexes ([GH2101](#))
- Panel.shift now supports negative periods ([GH2164](#))
- DataFrame now support unary `~` operator ([GH2110](#))

### 1.31.2 API changes

- Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window

```
In [1]: prng = period_range('2012Q1', periods=2, freq='Q')

In [2]: s = Series(np.random.randn(len(prng)), prng)

In [4]: s.resample('M')
Out [4]:
```

2012-01	-1.471992
2012-02	NaN
2012-03	NaN
2012-04	-0.493593

(continues on next page)

(continued from previous page)

```

2012-05      NaN
2012-06      NaN
Freq: M, dtype: float64

```

- `Period.end_time` now returns the last nanosecond in the time interval ([GH2124](#), [GH2125](#), [GH1764](#))

```

In [18]: p = Period('2012')

In [19]: p.end_time
Out[19]: Timestamp('2012-12-31 23:59:59.999999999')

```

- File parsers no longer coerce to float or bool for columns that have custom converters specified ([GH2184](#))

```

In [20]: data = 'A,B,C\n00001,001,5\n00002,002,6'

In [21]: read_csv(StringIO(data), converters={'A' : lambda x: x.strip()})
Out[21]:
   A  B  C
0  00001  1  5
1  00002  2  6

[2 rows x 3 columns]

```

See the [full release notes](#) or issue tracker on GitHub for a complete list.

## 1.32 v0.9.0 (October 7, 2012)

This is a major release from 0.8.1 and includes several new features and enhancements along with a large number of bug fixes. New features include vectorized unicode encoding/decoding for `Series.str`, `to_latex` method to `DataFrame`, more flexible parsing of boolean values, and enabling the download of options data from Yahoo! Finance.

### 1.32.1 New features

- Add `encode` and `decode` for unicode handling to *vectorized string processing methods* in `Series.str` ([GH1706](#))
- Add `DataFrame.to_latex` method ([GH1735](#))
- Add convenient expanding window equivalents of all `rolling_*` ops ([GH1785](#))
- Add `Options` class to `pandas.io.data` for fetching options data from Yahoo! Finance ([GH1748](#), [GH1739](#))
- More flexible parsing of boolean values (Yes, No, TRUE, FALSE, etc) ([GH1691](#), [GH1295](#))
- Add `level` parameter to `Series.reset_index`
- `TimeSeries.between_time` can now select times across midnight ([GH1871](#))
- `Series` constructor can now handle generator as input ([GH1679](#))
- `DataFrame.dropna` can now take multiple axes (tuple/list) as input ([GH924](#))
- Enable `skip_footer` parameter in `ExcelFile.parse` ([GH1843](#))

### 1.32.2 API changes

- The default column names when `header=None` and no columns names passed to functions like `read_csv` has changed to be more Pythonic and amenable to attribute access:

```
In [1]: data = '0,0,1\n1,1,0\n0,1,0'

In [2]: df = read_csv(StringIO(data), header=None)

In [3]: df
Out[3]:
```

	0	1	2
0	0	0	1
1	1	1	0
2	0	1	0

```
[3 rows x 3 columns]
```

- Creating a Series from another Series, passing an index, will cause reindexing to happen inside rather than treating the Series like an ndarray. Technically improper usages like `Series(df[col1], index=df[col2])` that worked before “by accident” (this was never intended) will lead to all NA Series in some cases. To be perfectly clear:

```
In [4]: s1 = Series([1, 2, 3])

In [5]: s1
Out[5]:
```

0	1
1	2
2	3

```
Length: 3, dtype: int64

In [6]: s2 = Series(s1, index=['foo', 'bar', 'baz'])

In [7]: s2
Out[7]:
```

foo	NaN
bar	NaN
baz	NaN

```
Length: 3, dtype: float64
```

- Deprecated `day_of_year` API removed from `PeriodIndex`, use `dayofyear` ([GH1723](#))
- Don't modify NumPy suppress printoption to True at import time
- The internal HDF5 data arrangement for DataFrames has been transposed. Legacy files will still be readable by `HDFStore` ([GH1834](#), [GH1824](#))
- Legacy cruft removed: `pandas.stats.misc.quantileTS`
- Use ISO8601 format for Period repr: monthly, daily, and on down ([GH1776](#))
- Empty DataFrame columns are now created as object dtype. This will prevent a class of `TypeError`s that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) ([GH1783](#))
- Setting parts of DataFrame/Panel using `ix` now aligns input Series/DataFrame ([GH1630](#))
- `first` and `last` methods in `GroupBy` no longer drop non-numeric columns ([GH1809](#))

- Resolved inconsistencies in specifying custom NA values in text parser. `na_values` of type dict no longer override default NAs unless `keep_default_na` is set to false explicitly (GH1657)
- `DataFrame.dot` will not do data alignment, and also work with Series (GH1915)

See the [full release notes](#) or issue tracker on GitHub for a complete list.

## 1.33 v0.8.1 (July 22, 2012)

This release includes a few new features, performance enhancements, and over 30 bug fixes from 0.8.0. New features include notably NA friendly string processing functionality and a series of new plot types and options.

### 1.33.1 New features

- Add *vectorized string processing methods* accessible via `Series.str` (GH620)
- Add option to disable adjustment in EWMA (GH1584)
- *Radviz plot* (GH1566)
- *Parallel coordinates plot*
- *Bootstrap plot*
- Per column styles and secondary y-axis plotting (GH1559)
- New datetime converters millisecond plotting (GH1599)
- Add option to disable “sparse” display of hierarchical indexes (GH1538)
- Series/DataFrame’s `set_index` method can *append levels* to an existing Index/MultiIndex (GH1569, GH1577)

### 1.33.2 Performance improvements

- Improved implementation of rolling min and max (thanks to Bottleneck !)
- Add accelerated 'median' GroupBy option (GH1358)
- Significantly improve the performance of parsing ISO8601-format date strings with `DatetimeIndex` or `to_datetime` (GH1571)
- Improve the performance of GroupBy on single-key aggregations and use with Categorical types
- Significant datetime parsing performance improvements

## 1.34 v0.8.0 (June 29, 2012)

This is a major release from 0.7.3 and includes extensive work on the time series handling and processing infrastructure as well as a great deal of new functionality throughout the library. It includes over 700 commits from more than 20 distinct authors. Most pandas 0.7.3 and earlier users should not experience any issues upgrading, but due to the migration to the NumPy `datetime64` dtype, there may be a number of bugs and incompatibilities lurking. Lingering incompatibilities will be fixed ASAP in a 0.8.1 release if necessary. See the [full release notes](#) or issue tracker on GitHub for a complete list.

### 1.34.1 Support for non-unique indexes

All objects can now work with non-unique indexes. Data alignment / join operations work according to SQL join semantics (including, if application, index duplication in many-to-many joins)

### 1.34.2 NumPy datetime64 dtype and 1.6 dependency

Time series data are now represented using NumPy's datetime64 dtype; thus, pandas 0.8.0 now requires at least NumPy 1.6. It has been tested and verified to work with the development version (1.7+) of NumPy as well which includes some significant user-facing API changes. NumPy 1.6 also has a number of bugs having to do with nanosecond resolution data, so I recommend that you steer clear of NumPy 1.6's datetime64 API functions (though limited as they are) and only interact with this data using the interface that pandas provides.

See the end of the 0.8.0 section for a “porting” guide listing potential issues for users migrating legacy codebases from pandas 0.7 or earlier to 0.8.0.

Bug fixes to the 0.7.x series for legacy NumPy < 1.6 users will be provided as they arise. There will be no more further development in 0.7.x beyond bug fixes.

### 1.34.3 Time series changes and improvements

---

**Note:** With this release, legacy `scikits.timeseries` users should be able to port their code to use pandas.

---

---

**Note:** See [documentation](#) for overview of pandas timeseries API.

---

- New datetime64 representation **speeds up join operations and data alignment, reduces memory usage**, and improve serialization / deserialization performance significantly over `datetime.datetime`
- High performance and flexible **resample** method for converting from high-to-low and low-to-high frequency. Supports interpolation, user-defined aggregation functions, and control over how the intervals and result labeling are defined. A suite of high performance Cython/C-based resampling functions (including Open-High-Low-Close) have also been implemented.
- Revamp of *frequency aliases* and support for **frequency shortcuts** like ‘15min’, or ‘1h30min’
- New *DatetimeIndex class* supports both fixed frequency and irregular time series. Replaces now deprecated `DateRange` class
- New `PeriodIndex` and `Period` classes for representing *time spans* and performing **calendar logic**, including the *12 fiscal quarterly frequencies* `<timeseries.quarterly>`. This is a partial port of, and a substantial enhancement to, elements of the `scikits.timeseries` codebase. Support for conversion between `PeriodIndex` and `DatetimeIndex`
- New Timestamp data type subclasses *datetime.datetime*, providing the same interface while enabling working with nanosecond-resolution data. Also provides *easy time zone conversions*.
- Enhanced support for *time zones*. Add `tz_convert` and `tz_localize` methods to `TimeSeries` and `DataFrame`. All timestamps are stored as UTC; Timestamps from `DatetimeIndex` objects with time zone set will be localized to localtime. Time zone conversions are therefore essentially free. User needs to know very little about `pytz` library now; only time zone names as strings are required. Time zone-aware timestamps are equal if and only if their UTC timestamps match. Operations between time zone-aware time series with different time zones will result in a UTC-indexed time series.
- Time series **string indexing conveniences** / shortcuts: slice years, year and month, and index values with strings

- Enhanced time series **plotting**; adaptation of scikits.timeseries matplotlib-based plotting code
- New `date_range`, `bdate_range`, and `period_range` *factory functions*
- Robust **frequency inference** function `infer_freq` and `inferred_freq` property of `DatetimeIndex`, with option to infer frequency on construction of `DatetimeIndex`
- `to_datetime` function efficiently **parses array of strings** to `DatetimeIndex`. `DatetimeIndex` will parse array or list of strings to `datetime64`
- **Optimized** support for `datetime64-dtype` data in `Series` and `DataFrame` columns
- New `NaT` (Not-a-Time) type to represent **NA** in timestamp arrays
- Optimize `Series.asof` for looking up “as of” values for arrays of timestamps
- Milli, Micro, Nano date offset objects
- Can index time series with `datetime.time` objects to select all data at particular **time of day** (`TimeSeries.at_time`) or **between two times** (`TimeSeries.between_time`)
- Add *tshift* method for leading/lagging using the frequency (if any) of the index, as opposed to a naive lead/lag using `shift`

#### 1.34.4 Other new features

- New *cut* and `qcut` functions (like R’s `cut` function) for computing a categorical variable from a continuous variable by binning values either into value-based (`cut`) or quantile-based (`qcut`) bins
- Rename `Factor` to `Categorical` and add a number of usability features
- Add *limit* argument to `fillna/reindex`
- More flexible multiple function application in `GroupBy`, and can pass list (name, function) tuples to get result in particular order with given names
- Add flexible *replace* method for efficiently substituting values
- Enhanced *read\_csv/read\_table* for reading time series data and converting multiple columns to dates
- Add *comments* option to parser functions: `read_csv`, etc.
- Add *dayfirst* option to parser functions for parsing international DD/MM/YYYY dates
- Allow the user to specify the CSV reader *dialect* to control quoting etc.
- Handling *thousands* separators in `read_csv` to improve integer parsing.
- Enable unstacking of multiple levels in one shot. Alleviate `pivot_table` bugs (empty columns being introduced)
- Move to `klib`-based hash tables for indexing; better performance and less memory usage than Python’s `dict`
- Add `first`, `last`, `min`, `max`, and `prod` optimized `GroupBy` functions
- New *ordered\_merge* function
- Add flexible *comparison* instance methods `eq`, `ne`, `lt`, `gt`, etc. to `DataFrame`, `Series`
- Improve *scatter\_matrix* plotting function and add histogram or kernel density estimates to diagonal
- Add ‘*kde*’ plot option for density plots
- Support for converting `DataFrame` to R `data.frame` through `rpy2`
- Improved support for complex numbers in `Series` and `DataFrame`

- Add *pct\_change* method to all data structures
- Add *max\_colwidth* configuration option for DataFrame console output
- *Interpolate* Series values using index values
- Can select multiple columns from GroupBy
- Add *update* methods to Series/DataFrame for updating values in place
- Add *any* and *all* method to DataFrame

### 1.34.5 New plotting methods

Series.plot now supports a *secondary\_y* option:

```
In [1]: plt.figure()
Out[1]: <Figure size 640x480 with 0 Axes>

In [2]: fx['FR'].plot(style='g')
Out[2]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c376acfd0>

In [3]: fx['IT'].plot(style='k--', secondary_y=True)
Out[3]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3781cb38>
```

Vytautas Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, 'kde' is a new option:

```
In [4]: s = Series(np.concatenate((np.random.randn(1000),
...                               np.random.randn(1000) * 0.5 + 3)))

In [5]: plt.figure()
Out[5]: <Figure size 640x480 with 0 Axes>

In [6]: s.hist(normed=True, alpha=0.2)
Out[6]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c37c3a588>

In [7]: s.plot(kind='kde')
Out[7]: <matplotlib.axes._subplots.AxesSubplot at 0x1c37c3a588>
```

See [the plotting page](#) for much more.

### 1.34.6 Other API changes

- Deprecation of *offset*, *time\_rule*, and *timeRule* arguments names in time series functions. Warnings will be printed until pandas 0.9 or 1.0.

### 1.34.7 Potential porting issues for pandas <= 0.7.3 users

The major change that may affect you in pandas 0.8.0 is that time series indexes use NumPy's *datetime64* data type instead of *dtype=object* arrays of Python's built-in *datetime.datetime* objects. *DateRange* has been





(continued from previous page)

```

datetime.datetime(2000, 1, 10, 0, 0)], dtype=object)

In [20]: dt_array[5]
//////////
→datetime.datetime(2000, 1, 6, 0, 0)

```

matplotlib knows how to handle `datetime.datetime` but not `Timestamp` objects. While I recommend that you plot time series using `TimeSeries.plot`, you can either use `to_pydatetime` or register a converter for the `Timestamp` type. See [matplotlib documentation](#) for more on this.

**Warning:** There are bugs in the user-facing API with the nanosecond `datetime64` unit in NumPy 1.6. In particular, the string version of the array shows garbage values, and conversion to `dtype=object` is similarly broken.

```

In [21]: rng = date_range('1/1/2000', periods=10)

In [22]: rng
Out [22]:
DatetimeIndex(['2000-01-01', '2000-01-02', '2000-01-03', '2000-01-04',
               '2000-01-05', '2000-01-06', '2000-01-07', '2000-01-08',
               '2000-01-09', '2000-01-10'],
              dtype='datetime64[ns]', freq='D')

In [23]: np.asarray(rng)
//////////
→
array(['2000-01-01T00:00:00.000000000', '2000-01-02T00:00:00.000000000',
       '2000-01-03T00:00:00.000000000', '2000-01-04T00:00:00.000000000',
       '2000-01-05T00:00:00.000000000', '2000-01-06T00:00:00.000000000',
       '2000-01-07T00:00:00.000000000', '2000-01-08T00:00:00.000000000',
       '2000-01-09T00:00:00.000000000', '2000-01-10T00:00:00.000000000'], dtype=
→'datetime64[ns]')

In [24]: converted = np.asarray(rng, dtype=object)

In [25]: converted[5]
Out [25]: 947116800000000000

```

**Trust me: don't panic.** If you are using NumPy 1.6 and restrict your interaction with `datetime64` values to pandas's API you will be just fine. There is nothing wrong with the data-type (a 64-bit integer internally); all of the important data processing happens in pandas and is heavily tested. I strongly recommend that you **do not work directly with `datetime64` arrays in NumPy 1.6** and only use the pandas API.

**Support for non-unique indexes:** In the latter case, you may have code inside a `try:... catch:` block that failed due to the index not being unique. In many cases it will no longer fail (some method like `append` still check for uniqueness unless disabled). However, all is not lost: you can inspect `index.is_unique` and raise an exception explicitly if it is `False` or go to a different code branch.

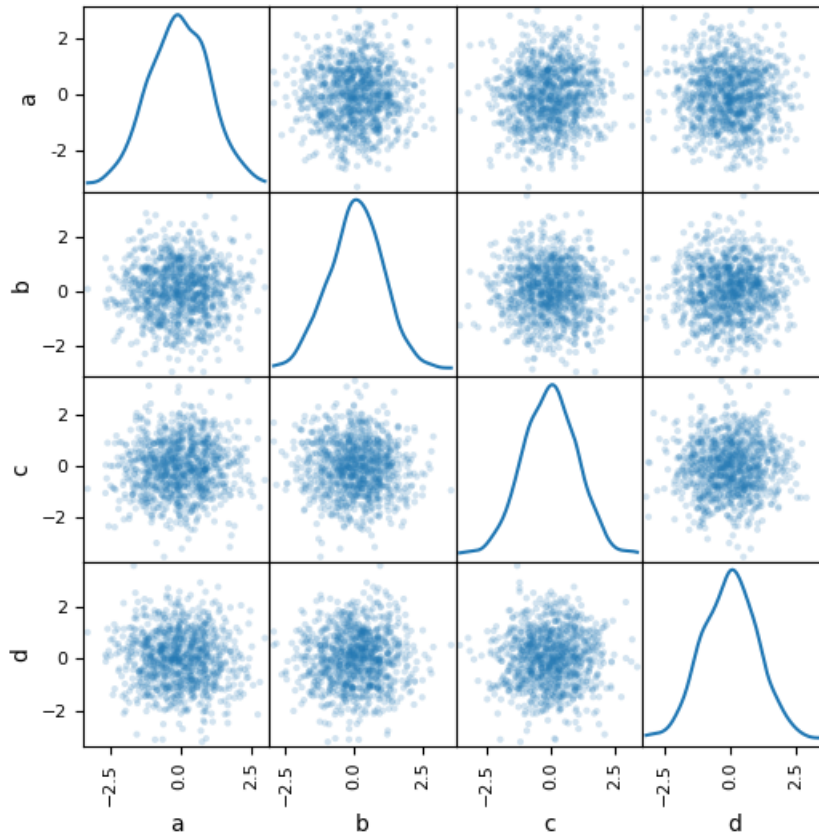
## 1.35 v.0.7.3 (April 12, 2012)

This is a minor release from 0.7.2 and fixes many minor bugs and adds a number of nice new features. There are also a couple of API changes to note; these should not affect very many users, and we are inclined to call them “bug fixes” even though they do constitute a change in behavior. See the [full release notes](#) or issue tracker on GitHub for a complete list.

### 1.35.1 New features

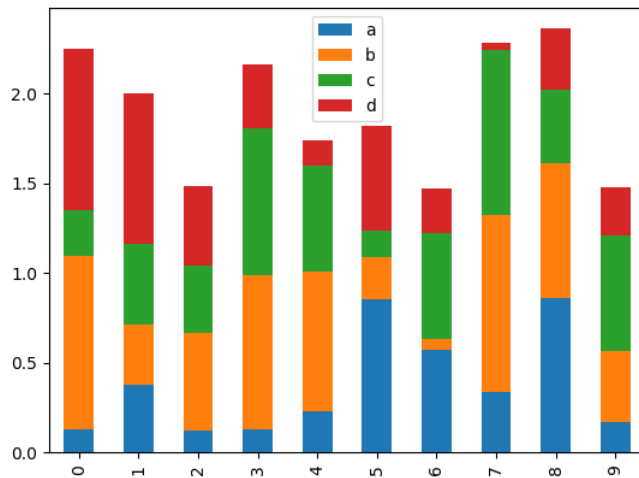
- New *fixed width file reader*, `read_fwf`
- New *scatter\_matrix* function for making a scatter plot matrix

```
from pandas.tools.plotting import scatter_matrix
scatter_matrix(df, alpha=0.2)
```

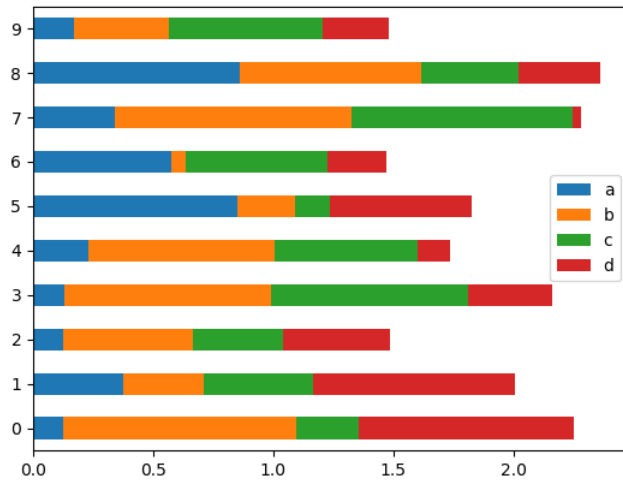


- Add `stacked` argument to `Series` and `DataFrame`'s `plot` method for *stacked bar plots*.

```
df.plot(kind='bar', stacked=True)
```



```
df.plot(kind='barh', stacked=True)
```



- Add log x and y *scaling options* to `DataFrame.plot` and `Series.plot`
- Add kurt methods to `Series` and `DataFrame` for computing kurtosis

### 1.35.2 NA Boolean Comparison API Change

Reverted some changes to how NA values (represented typically as `NaN` or `None`) are handled in non-numeric `Series`:

```
In [1]: series = Series(['Steve', np.nan, 'Joe'])
In [2]: series == 'Steve'
Out[2]:
0      True
1     False
```

(continues on next page)

(continued from previous page)

```

2    False
Length: 3, dtype: bool

In [3]: series != 'Steve'
Out[3]:
0    False
1     True
2     True
Length: 3, dtype: bool

```

In comparisons, NA / NaN will always come through as `False` except with `!=` which is `True`. *Be very careful* with boolean arithmetic, especially negation, in the presence of NA data. You may wish to add an explicit NA filter into boolean array operations if you are worried about this:

```

In [4]: mask = series == 'Steve'

In [5]: series[mask & series.notnull()]
Out[5]:
0    Steve
Length: 1, dtype: object

```

While propagating NA in comparisons may seem like the right behavior to some users (and you could argue on purely technical grounds that this is the right thing to do), the evaluation was made that propagating NA everywhere, including in numerical arrays, would cause a large amount of problems for users. Thus, a “practicality beats purity” approach was taken. This issue may be revisited at some point in the future.

### 1.35.3 Other API Changes

When calling `apply` on a grouped Series, the return value will also be a Series, to be more consistent with the groupby behavior with DataFrame:

```

In [6]: df = DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
...:                          'foo', 'bar', 'foo', 'foo'],
...:                   'B' : ['one', 'one', 'two', 'three',
...:                          'two', 'two', 'one', 'three'],
...:                   'C' : np.random.randn(8), 'D' : np.random.randn(8)})

In [7]: df
Out[7]:
   A    B         C         D
0  foo  one -0.841015  0.459840
1  bar  one  0.114219 -0.253040
2  foo  two -0.405617 -0.261128
3  bar three  1.240678  0.406604
4  foo  two -0.122828 -1.022256
5  bar  two  1.525196 -0.882785
6  foo  one  0.520047  1.793331
7  foo three  0.163834 -0.429688

[8 rows x 4 columns]

In [8]: grouped = df.groupby('A')['C']

In [9]: grouped.describe()

```

(continues on next page)

(continued from previous page)

```

Out [9]:
   count    mean    std    min    25%    50%    75%    max
A
bar     3.0  0.960031  0.746181  0.114219  0.677448  1.240678  1.525196
foo     5.0 -0.137116  0.522064 -0.841015 -0.405617 -0.122828  0.163834  0.520047

[2 rows x 8 columns]

In [10]: grouped.apply(lambda x: x.sort_values()[-2:]) # top 2 values
\
↪
A
bar  3    1.240678
     5    1.525196
foo  7    0.163834
     6    0.520047
Name: C, Length: 4, dtype: float64

```

## 1.36 v.0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

### 1.36.1 New features

- Add additional tie-breaking methods in `DataFrame.rank` (GH874)
- Add ascending parameter to rank in Series, DataFrame (GH875)
- Add `coerce_float` option to `DataFrame.from_records` (GH893)
- Add `sort_columns` parameter to allow unsorted plots (GH918)
- Enable column access via attributes on `GroupBy` (GH882)
- Can pass dict of values to `DataFrame.fillna` (GH661)
- Can select multiple hierarchical groups by passing list of values in `.ix` (GH134)
- Add `axis` option to `DataFrame.fillna` (GH174)
- Add `level` keyword to `drop` for dropping values from a level (GH159)

### 1.36.2 Performance improvements

- Use khash for `Series.value_counts`, add raw function to `algorithms.py` (GH861)
- Intercept `__builtin__.sum` in `groupby` (GH885)

## 1.37 v.0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

### 1.37.1 New features

- Add `to_clipboard` function to pandas namespace for writing objects to the system clipboard ([GH774](#))
- Add `itertuples` method to `DataFrame` for iterating through the rows of a dataframe as tuples ([GH818](#))
- Add ability to pass `fill_value` and method to `DataFrame` and `Series` align method ([GH806](#), [GH807](#))
- Add `fill_value` option to `reindex`, align methods ([GH784](#))
- Enable `concat` to produce `DataFrame` from `Series` ([GH787](#))
- Add `between` method to `Series` ([GH802](#))
- Add HTML representation hook to `DataFrame` for the IPython HTML notebook ([GH773](#))
- Support for reading Excel 2007 XML documents using `openpyxl`

### 1.37.2 Performance improvements

- Improve performance and memory usage of `fillna` on `DataFrame`
- Can concatenate a list of `Series` along `axis=1` to obtain a `DataFrame` ([GH787](#))

## 1.38 v.0.7.0 (February 9, 2012)

### 1.38.1 New features

- New unified *merge function* for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains ([GH220](#), [GH249](#), [GH267](#))
- New *unified concatenation function* for concatenating `Series`, `DataFrame` or `Panel` objects along an axis. Can form union or intersection of the other axes. Improves performance of `Series.append` and `DataFrame.append` ([GH468](#), [GH479](#), [GH273](#))
- *Can* pass multiple `DataFrames` to `DataFrame.append` to concatenate (stack) and multiple `Series` to `Series.append` too
- *Can* pass list of dicts (e.g., a list of JSON objects) to `DataFrame` constructor ([GH526](#))
- You can now *set multiple columns* in a `DataFrame` via `__getitem__`, useful for transformation ([GH342](#))
- Handle differently-indexed output values in `DataFrame.apply` ([GH498](#))

```
In [1]: df = DataFrame(randn(10, 4))

In [2]: df.apply(lambda x: x.describe())
Out[2]:
```

	0	1	2	3
count	10.000000	10.000000	10.000000	10.000000
mean	0.424980	0.115056	0.452000	-0.103829
std	0.898046	0.712034	0.867316	0.830197
min	-1.337024	-0.779344	-1.206466	-1.022360
25%	-0.000996	-0.344789	-0.120217	-0.744313
50%	0.860419	0.098422	0.540168	-0.387554
75%	1.094236	0.375137	1.076331	0.674952
max	1.207725	1.601703	1.663859	1.096187

(continues on next page)

(continued from previous page)

[8 rows x 4 columns]

- *Add* `reorder_levels` method to `Series` and `DataFrame` (GH534)
- *Add* dict-like `get` function to `DataFrame` and `Panel` (GH521)
- *Add* `DataFrame.iterrows` method for efficiently iterating through the rows of a `DataFrame`
- *Add* `DataFrame.to_panel` with code adapted from `LongPanel.to_long`
- *Add* `reindex_axis` method added to `DataFrame`
- *Add* `level` option to binary arithmetic functions on `DataFrame` and `Series`
- *Add* `level` option to the `reindex` and `align` methods on `Series` and `DataFrame` for broadcasting values across a level (GH542, GH552, others)
- *Add* attribute-based item access to `Panel` and add IPython completion (GH563)
- *Add* `logy` option to `Series.plot` for log-scaling on the Y axis
- *Add* `index` and `header` options to `DataFrame.to_string`
- *Can* pass multiple `DataFrames` to `DataFrame.join` to join on index (GH115)
- *Can* pass multiple `Panels` to `Panel.join` (GH115)
- *Added* `justify` argument to `DataFrame.to_string` to allow different alignment of column headers
- *Add* `sort` option to `GroupBy` to allow disabling sorting of the group keys for potential speedups (GH595)
- *Can* pass `MaskedArray` to `Series` constructor (GH563)
- *Add* `Panel` item access via attributes and IPython completion (GH554)
- Implement `DataFrame.lookup`, fancy-indexing analogue for retrieving values given a sequence of row and column labels (GH338)
- Can pass a *list of functions* to aggregate with `groupby` on a `DataFrame`, yielding an aggregated result with hierarchical columns (GH166)
- Can call `cummin` and `cummax` on `Series` and `DataFrame` to get cumulative minimum and maximum, respectively (GH647)
- `value_range` added as utility function to get min and max of a dataframe (GH288)
- Added `encoding` argument to `read_csv`, `read_table`, `to_csv` and `from_csv` for non-ascii text (GH717)
- *Added* `abs` method to pandas objects
- *Added* `crosstab` function for easily computing frequency tables
- *Added* `isin` method to index objects
- *Added* `level` argument to `xs` method of `DataFrame`.

### 1.38.2 API Changes to integer indexing

One of the potentially riskiest API changes in 0.7.0, but also one of the most important, was a complete review of how **integer indexes** are handled with regard to label-based indexing. Here is an example:



```
In [3]: s = Series(randn(10), index=range(0, 20, 2))
```

In [4]: s

Out [4] :

```
0      0.446246
2     -0.500268
4      0.814725
6     -0.312744
8      1.098892
10     1.306330
12    -0.366970
14    -0.030890
16     1.608095
18    -0.023287
Length: 10, dtype: float64
```

```
In [5]: s[0]
```

0.44624598505731339

```
In [6]: s[2]
```

$\rightarrow -0.500268093241102$

```
In [7]: s[4]
```

0.8147247587659604

This is all exactly identical to the behavior before. However, if you ask for a key **not** contained in the Series, in versions 0.6.1 and prior, Series would *fall back* on a location-based lookup. This now raises a `KeyError`:

```
In [2]: s[1]
```

```
KeyError: 1
```

This change also has the same impact on DataFrame:

```
In [3]: df = DataFrame(randn(8, 4), index=range(0, 16, 2))
```

```
In [4]: df
```

	0	1	2	3
0	0.88427	0.3363	-0.1787	0.03162
2	0.14451	-0.1415	0.2504	0.58374
4	-1.44779	-0.9186	-1.4996	0.27163
6	-0.26598	-2.4184	-0.2658	0.11503
8	-0.58776	0.3144	-0.8566	0.61941
10	0.10940	-0.7175	-1.0108	0.47990
12	-1.16919	-0.3087	-0.6049	-0.43544
14	-0.07337	0.3410	0.0424	-0.16037

```
In [5]: df.ix[3]
```

```
KeyError: 3
```

In order to support purely integer-based indexing, the following methods have been added:

Method	Description
<code>Series.iat_value(i)</code>	Retrieve value stored at location <i>i</i>
<code>Series.iat(i)</code>	Alias for <code>iat_value</code>
<code>DataFrame.irow(i)</code>	Retrieve the <i>i</i> -th row
<code>DataFrame.icol(j)</code>	Retrieve the <i>j</i> -th column
<code>DataFrame.iat_value(i, j)</code>	Retrieve the value at row <i>i</i> and column <i>j</i>

### 1.38.3 API tweaks regarding label-based slicing

Label-based slicing using `ix` now requires that the index be sorted (monotonic) **unless** both the start and endpoint are contained in the index:

```
In [1]: s = Series(randn(6), index=list('gmkaec'))

In [2]: s
Out[2]:
g   -1.182230
m   -0.276183
k   -0.243550
a    1.628992
e    0.073308
c   -0.539890
dtype: float64
```

Then this is OK:

```
In [3]: s.ix['k':'e']
Out[3]:
k   -0.243550
a    1.628992
e    0.073308
dtype: float64
```

But this is not:

```
In [12]: s.ix['b':'h']
KeyError 'b'
```

If the index had been sorted, the “range selection” would have been possible:

```
In [4]: s2 = s.sort_index()

In [5]: s2
Out[5]:
a    1.628992
c   -0.539890
e    0.073308
g   -1.182230
k   -0.243550
m   -0.276183
dtype: float64

In [6]: s2.ix['b':'h']
Out[6]:
c   -0.539890
```

(continues on next page)

(continued from previous page)

```
e    0.073308
g   -1.182230
dtype: float64
```

### 1.38.4 Changes to Series [] operator

As as notational convenience, you can pass a sequence of labels or a label slice to a Series when getting and setting values via [] (i.e. the `__getitem__` and `__setitem__` methods). The behavior will be the same as passing similar input to `ix` **except in the case of integer indexing**:

```
In [8]: s = Series(randn(6), index=list('acegkm'))
```

```
In [9]: s
```

```
Out[9]:
```

```
a    -0.800734
c    -0.229737
e    -0.781940
g     0.756053
k     2.613373
m    -0.159310
Length: 6, dtype: float64
```

```
In [10]: s[['m', 'a', 'c', 'e']]
```

```
////////////////////////////////////
↪
m    -0.159310
a    -0.800734
c    -0.229737
e    -0.781940
Length: 4, dtype: float64
```

```
In [11]: s['b':'l']
```

```
////////////////////////////////////
↪
c    -0.229737
e    -0.781940
g     0.756053
k     2.613373
Length: 4, dtype: float64
```

```
In [12]: s['c':'k']
```

```
////////////////////////////////////
↪
c    -0.229737
e    -0.781940
g     0.756053
k     2.613373
Length: 4, dtype: float64
```

In the case of integer indexes, the behavior will be exactly as before (shadowing `ndarray`):

```
In [13]: s = Series(randn(6), index=range(0, 12, 2))
```

```
In [14]: s[[4, 0, 2]]
```

```
Out[14]:
```

(continues on next page)

(continued from previous page)

```

4      0.022862
0     -0.321246
2     -0.707337
Length: 3, dtype: float64

In [15]: s[1:5]
Out[15]:
2     -0.707337
4      0.022862
6      0.306713
8     -0.162222
Length: 4, dtype: float64

```

If you wish to do indexing with sequences and slicing on an integer index with label semantics, use `ix`.

### 1.38.5 Other API Changes

- The deprecated `LongPanel` class has been completely removed
- If `Series.sort` is called on a column of a `DataFrame`, an exception will now be raised. Before it was possible to accidentally mutate a `DataFrame`'s column by doing `df[col].sort()` instead of the side-effect free method `df[col].order()` ([GH316](#))
- Miscellaneous renames and deprecations which will (harmlessly) raise `FutureWarning`
- `drop` added as an optional parameter to `DataFrame.reset_index` ([GH699](#))

### 1.38.6 Performance improvements

- *Cythonized GroupBy aggregations* no longer presort the data, thus achieving a significant speedup ([GH93](#)). `GroupBy` aggregations with Python functions significantly sped up by clever manipulation of the `ndarray` data type in Cython ([GH496](#)).
- Better error message in `DataFrame` constructor when passed column labels don't match data ([GH497](#))
- Substantially improve performance of multi-`GroupBy` aggregation when a Python function is passed, reuse `ndarray` object in Cython ([GH496](#))
- Can store objects indexed by tuples and floats in `HDFStore` ([GH492](#))
- Don't print length by default in `Series.to_string`, add `length` option ([GH489](#))
- Improve Cython code for multi-groupby to aggregate without having to sort the data ([GH93](#))
- Improve `MultiIndex` reindexing speed by storing tuples in the `MultiIndex`, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of `Series.__getitem__` for standard use cases
- Avoid `Index` dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in `setup.py` if `NumPy` not installed
- Use common set of NA-handling operations (sum, mean, etc.) in `Panel` class also ([GH536](#))
- Default name assignment when calling `reset_index` on `DataFrame` with a regular (non-hierarchical) index ([GH476](#))

- Use Cythonized groupers when possible in Series/DataFrame stat ops with `level` parameter passed (GH545)
- Ported skiplist data structure to C to speed up `rolling_median` by about 5-10x in most typical use cases (GH374)

## 1.39 v.0.6.1 (December 13, 2011)

### 1.39.1 New features

- Can *append single rows* (as Series) to a DataFrame
- Add Spearman and Kendall rank *correlation* options to `Series.corr` and `DataFrame.corr` (GH428)
- *Added* `get_value` and `set_value` methods to Series, DataFrame, and Panel for very low-overhead access (>2x faster in many cases) to scalar elements (GH437, GH438). `set_value` is capable of producing an enlarged object.
- Add PyQt table widget to sandbox (GH435)
- `DataFrame.align` can *accept Series arguments* and an *axis option* (GH461)
- Implement new *SparseArray* and *SparseList* data structures. `SparseSeries` now derives from `SparseArray` (GH463)
- *Better console printing options* (GH453)
- Implement fast *data ranking* for Series and DataFrame, fast versions of `scipy.stats.rankdata` (GH428)
- Implement *DataFrame.from\_items* alternate constructor (GH444)
- `DataFrame.convert_objects` method for *inferring better dtypes* for object columns (GH302)
- Add *rolling\_corr\_pairwise* function for computing Panel of correlation matrices (GH189)
- Add *margins* option to *pivot\_table* for computing subgroup aggregates (GH114)
- Add `Series.from_csv` function (GH482)
- *Can pass* DataFrame/DataFrame and DataFrame/Series to `rolling_corr/rolling_cov` (GH #462)
- `MultiIndex.get_level_values` can *accept the level name*

### 1.39.2 Performance improvements

- Improve memory usage of *DataFrame.describe* (do not copy data unnecessarily) (PR #425)
- Optimize scalar value lookups in the general case by 25% or more in Series and DataFrame
- Fix performance regression in cross-sectional count in DataFrame, affecting `DataFrame.dropna` speed
- Column deletion in DataFrame copies no data (computes views on blocks) (GH #158)

## 1.40 v.0.6.0 (November 25, 2011)

### 1.40.1 New Features

- *Added* `melt` function to `pandas.core.reshape`
- *Added* `level` parameter to `group by level` in Series and DataFrame descriptive statistics (GH313)

- *Added* `head` and `tail` methods to `Series`, analogous to `DataFrame` (GH296)
- *Added* `Series.isin` function which checks if each value is contained in a passed sequence (GH289)
- *Added* `float_format` option to `Series.to_string`
- *Added* `skip_footer` (GH291) and `converters` (GH343) options to `read_csv` and `read_table`
- *Added* `drop_duplicates` and `duplicated` functions for removing duplicate `DataFrame` rows and checking for duplicate rows, respectively (GH319)
- *Implemented* operators `&`, `|`, `^`, `-` on `DataFrame` (GH347)
- *Added* `Series.mad`, mean absolute deviation
- *Added* `QuarterEnd` `DateOffset` (GH321)
- *Added* `dot` to `DataFrame` (GH65)
- *Added* `orient` option to `Panel.from_dict` (GH359, GH301)
- *Added* `orient` option to `DataFrame.from_dict`
- *Added* passing list of tuples or list of lists to `DataFrame.from_records` (GH357)
- *Added* multiple levels to `groupby` (GH103)
- *Allow* multiple columns in `by` argument of `DataFrame.sort_index` (GH92, GH362)
- *Added* `fast_get_value` and `put_value` methods to `DataFrame` (GH360)
- *Added* `cov` instance methods to `Series` and `DataFrame` (GH194, GH362)
- *Added* `kind='bar'` option to `DataFrame.plot` (GH348)
- *Added* `idxmin` and `idxmax` to `Series` and `DataFrame` (GH286)
- *Added* `read_clipboard` function to parse `DataFrame` from clipboard (GH300)
- *Added* `nunique` function to `Series` for counting unique elements (GH297)
- *Made* `DataFrame` constructor use `Series` name if no columns passed (GH373)
- *Support* regular expressions in `read_table/read_csv` (GH364)
- *Added* `DataFrame.to_html` for writing `DataFrame` to HTML (GH387)
- *Added* support for `MaskedArray` data in `DataFrame`, masked values converted to `NaN` (GH396)
- *Added* `DataFrame.boxplot` function (GH368)
- *Can* pass extra args, kwds to `DataFrame.apply` (GH376)
- *Implement* `DataFrame.join` with vector on argument (GH312)
- *Added* `legend` boolean flag to `DataFrame.plot` (GH324)
- *Can* pass multiple levels to `stack` and `unstack` (GH370)
- *Can* pass multiple values columns to `pivot_table` (GH381)
- *Use* `Series` name in `GroupBy` for result index (GH363)
- *Added* `raw` option to `DataFrame.apply` for performance if only need `ndarray` (GH309)
- *Added* proper, tested weighted least squares to standard and panel OLS (GH303)

### 1.40.2 Performance Enhancements

- VBENCH Cythonized `cache_readonly`, resulting in substantial micro-performance enhancements throughout the codebase ([GH361](#))
- VBENCH Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than `np.apply_along_axis` ([GH309](#))
- VBENCH Improved performance of `MultiIndex.from_tuples`
- VBENCH Special Cython matrix iterator for applying arbitrary reduction operations
- VBENCH + DOCUMENT Add `raw` option to `DataFrame.apply` for getting better performance when
- VBENCH Faster cythonized count by level in `Series` and `DataFrame` ([GH341](#))
- VBENCH? Significant `GroupBy` performance enhancement with multiple keys with many “empty” combinations
- VBENCH New Cython vectorized function `map_infer` speeds up `Series.apply` and `Series.map` significantly when passed elementwise Python function, motivated by ([GH355](#))
- VBENCH Significantly improved performance of `Series.order`, which also makes `np.unique` called on a `Series` faster ([GH327](#))
- VBENCH Vastly improved performance of `GroupBy` on axes with a `MultiIndex` ([GH299](#))

## 1.41 v.0.5.0 (October 24, 2011)

### 1.41.1 New Features

- *Added* `DataFrame.align` method with standard join options
- *Added* `parse_dates` option to `read_csv` and `read_table` methods to optionally try to parse dates in the index columns
- *Added* `nrows`, `chunksize`, and `iterator` arguments to `read_csv` and `read_table`. The last two return a new `TextParser` class capable of lazily iterating through chunks of a flat file ([GH242](#))
- *Added* ability to join on multiple columns in `DataFrame.join` ([GH214](#))
- Added private `_get_duplicates` function to `Index` for identifying duplicate values more easily ([ENH5c](#))
- *Added* column attribute access to `DataFrame`.
- *Added* Python tab completion hook for `DataFrame` columns. ([GH233](#), [GH230](#))
- *Implemented* `Series.describe` for `Series` containing objects ([GH241](#))
- *Added* inner join option to `DataFrame.join` when joining on key(s) ([GH248](#))
- *Implemented* selecting `DataFrame` columns by passing a list to `__getitem__` ([GH253](#))
- *Implemented* `&` and `|` to intersect / union `Index` objects, respectively ([GH261](#))
- *Added* `pivot_table` convenience function to pandas namespace ([GH234](#))
- *Implemented* `Panel.rename_axis` function ([GH243](#))
- `DataFrame` will show index level names in console output ([GH334](#))
- *Implemented* `Panel.take`
- *Added* `set_eng_float_format` for alternate `DataFrame` floating point string formatting ([ENH61](#))

- *Added* convenience `set_index` function for creating a `DataFrame` index from its existing columns
- *Implemented* `groupby` hierarchical index level name (GH223)
- *Added* support for different delimiters in `DataFrame.to_csv` (GH244)
- TODO: DOCS ABOUT TAKE METHODS

### 1.41.2 Performance Enhancements

- VBENCH Major performance improvements in file parsing functions `read_csv` and `read_table`
- VBENCH Added Cython function for converting tuples to `ndarray` very fast. Speeds up many `MultiIndex`-related operations
- VBENCH Refactored merging / joining code into a tidy class and disabled unnecessary computations in the float/object case, thus getting about 10% better performance (GH211)
- VBENCH Improved speed of `DataFrame.xs` on mixed-type `DataFrame` objects by about 5x, regression from 0.3.0 (GH215)
- VBENCH With new `DataFrame.align` method, speeding up binary operations between differently-indexed `DataFrame` objects by 10-25%.
- VBENCH Significantly sped up conversion of nested dict into `DataFrame` (GH212)
- VBENCH Significantly speed up `DataFrame.__repr__` and `count` on large mixed-type `DataFrame` objects

## 1.42 v.0.4.3 through v0.4.1 (September 25 - October 9, 2011)

### 1.42.1 New Features

- Added Python 3 support using 2to3 (GH200)
- *Added* name attribute to `Series`, now prints as part of `Series.__repr__`
- *Added* instance methods `isnull` and `notnull` to `Series` (GH209, GH203)
- *Added* `Series.align` method for aligning two series with choice of join method (ENH56)
- *Added* method `get_level_values` to `MultiIndex` (GH188)
- Set values in mixed-type `DataFrame` objects via `.ix` indexing attribute (GH135)
- Added new `DataFrame` *methods* `get_dtype_counts` and property `dtypes` (ENHdc)
- Added *ignore\_index* option to `DataFrame.append` to stack `DataFrames` (ENH1b)
- `read_csv` tries to *sniff* delimiters using `csv.Sniffer` (GH146)
- `read_csv` can *read* multiple columns into a `MultiIndex`; `DataFrame`'s `to_csv` method writes out a corresponding `MultiIndex` (GH151)
- `DataFrame.rename` has a new `copy` parameter to *rename* a `DataFrame` in place (ENHed)
- *Enable* unstacking by name (GH142)
- *Enable* `sortlevel` to work by level (GH141)



### 1.42.2 Performance Enhancements

- Altered binary operations on differently-indexed SparseSeries objects to use the integer-based (dense) alignment logic which is faster with a larger number of blocks ([GH205](#))
- Wrote faster Cython data alignment / merging routines resulting in substantial speed increases
- Improved performance of `isnull` and `notnull`, a regression from v0.3.0 ([GH187](#))
- Refactored code related to `DataFrame.join` so that intermediate aligned copies of the data in each `DataFrame` argument do not need to be created. Substantial performance increases result ([GH176](#))
- Substantially improved performance of generic `Index.intersection` and `Index.union`
- Implemented `BlockManager.take` resulting in significantly faster `take` performance on mixed-type `DataFrame` objects ([GH104](#))
- Improved performance of `Series.sort_index`
- Significant groupby performance enhancement: removed unnecessary integrity checks in `DataFrame` internals that were slowing down slicing operations to retrieve groups
- Optimized `_ensure_index` function resulting in performance savings in type-checking `Index` objects
- Wrote fast time series merging / joining methods in Cython. Will be integrated later into `DataFrame.join` and related functions



## INSTALLATION

The easiest way to install pandas is to install it as part of the [Anaconda](#) distribution, a cross platform distribution for data analysis and scientific computing. This is the recommended installation method for most users.

Instructions for installing from source, [PyPI](#), [ActivePython](#), various Linux distributions, or a [development version](#) are also provided.

### 2.1 Plan for dropping Python 2.7

The Python core team plans to stop supporting Python 2.7 on January 1st, 2020. In line with [NumPy's plans](#), all pandas releases through December 31, 2018 will support Python 2.

The final release before **December 31, 2018** will be the last release to support Python 2. The released package will continue to be available on PyPI and through conda.

Starting **January 1, 2019**, all releases will be Python 3 only.

If there are people interested in continued support for Python 2.7 past December 31, 2018 (either backporting bugfixes or funding) please reach out to the maintainers on the issue tracker.

For more information, see the [Python 3 statement](#) and the [Porting to Python 3 guide](#).

### 2.2 Python version support

Officially Python 2.7, 3.5, 3.6, and 3.7.

### 2.3 Installing pandas

#### 2.3.1 Installing with Anaconda

Installing pandas and the rest of the [NumPy](#) and [SciPy](#) stack can be a little difficult for inexperienced users.

The simplest way to install not only pandas, but Python and the most popular packages that make up the [SciPy](#) stack ([IPython](#), [NumPy](#), [Matplotlib](#), ...) is with [Anaconda](#), a cross-platform (Linux, Mac OS X, Windows) Python distribution for data analytics and scientific computing.

After running the installer, the user will have access to pandas and the rest of the [SciPy](#) stack without needing to install anything else, and without needing to wait for any software to be compiled.

Installation instructions for [Anaconda](#) can be found [here](#).

A full list of the packages available as part of the [Anaconda](#) distribution [can be found here](#).

Another advantage to installing Anaconda is that you don't need admin rights to install it. Anaconda can install in the user's home directory, which makes it trivial to delete Anaconda if you decide (just delete that folder).

## 2.3.2 Installing with Miniconda

The previous section outlined how to get pandas installed as part of the [Anaconda](#) distribution. However this approach means you will install well over one hundred packages and involves downloading the installer which is a few hundred megabytes in size.

If you want to have more control on which packages, or have a limited internet bandwidth, then installing pandas with [Miniconda](#) may be a better solution.

[Conda](#) is the package manager that the [Anaconda](#) distribution is built upon. It is a package manager that is both cross-platform and language agnostic (it can play a similar role to a pip and virtualenv combination).

[Miniconda](#) allows you to create a minimal self contained Python installation, and then use the [Conda](#) command to install additional packages.

First you will need [Conda](#) to be installed and downloading and running the [Miniconda](#) will do this for you. The installer [can be found here](#)

The next step is to create a new conda environment. A conda environment is like a virtualenv that allows you to specify a specific version of Python and set of libraries. Run the following commands from a terminal window:

```
conda create -n name_of_my_env python
```

This will create a minimal environment with only Python installed in it. To put your self inside this environment run:

```
source activate name_of_my_env
```

On Windows the command is:

```
activate name_of_my_env
```

The final step required is to install pandas. This can be done with the following command:

```
conda install pandas
```

To install a specific pandas version:

```
conda install pandas=0.20.3
```

To install other packages, IPython for example:

```
conda install ipython
```

To install the full [Anaconda](#) distribution:

```
conda install anaconda
```

If you need packages that are available to pip but not conda, then install pip, and then use pip to install those packages:

```
conda install pip
pip install django
```

### 2.3.3 Installing from PyPI

pandas can be installed via pip from PyPI.

```
pip install pandas
```

### 2.3.4 Installing with ActivePython

Installation instructions for [ActivePython](#) can be found [here](#). Versions 2.7 and 3.5 include pandas.

### 2.3.5 Installing using your Linux distribution's package manager.

The commands in this table will install pandas for Python 3 from your distribution. To install pandas for Python 2, you may need to use the `python-pandas` package.

Distribution	Status	Download / Repository Link	Install method
Debian	stable	<a href="#">official Debian repository</a>	<code>sudo apt-get install python3-pandas</code>
Debian & Ubuntu	unstable (latest packages)	<a href="#">NeuroDebian</a>	<code>sudo apt-get install python3-pandas</code>
Ubuntu	stable	<a href="#">official Ubuntu repository</a>	<code>sudo apt-get install python3-pandas</code>
OpenSuse	stable	<a href="#">OpenSuse Repository</a>	<code>zypper in python3-pandas</code>
Fedora	stable	<a href="#">official Fedora repository</a>	<code>dnf install python3-pandas</code>
Centos/RHEL	stable	<a href="#">EPEL repository</a>	<code>yum install python3-pandas</code>

**However**, the packages in the linux package managers are often a few versions behind, so to get the newest version of pandas, it's recommended to install using the `pip` or `conda` methods described above.

### 2.3.6 Installing from source

See the [contributing documentation](#) for complete instructions on building from the git source tree. Further, see [creating a development environment](#) if you wish to create a *pandas* development environment.

## 2.4 Running the test suite

pandas is equipped with an exhaustive set of unit tests, covering about 97% of the codebase as of this writing. To run it on your machine to verify that everything is working (and that you have all of the dependencies, soft and hard, installed), make sure you have `pytest` and run:

```
>>> import pandas as pd
>>> pd.test()
running: pytest --skip-slow --skip-network C:\Users\TP\Anaconda3\envs\py36\lib\site-
packages\pandas
===== test session starts =====
```

(continues on next page)

(continued from previous page)

```
platform win32 -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: C:\Users\TP\Documents\Python\pandasdev\pandas, inifile: setup.cfg
collected 12145 items / 3 skipped

.....S.....
.....S.....
.....

===== 12130 passed, 12 skipped in 368.339 seconds =====
```

## 2.5 Dependencies

- **setuptools**: 24.2.0 or higher
- **NumPy**: 1.9.0 or higher
- **python-dateutil**: 2.5.0 or higher
- **pytz**

### 2.5.1 Recommended Dependencies

- **numexpr**: for accelerating certain numerical operations. **numexpr** uses multiple cores as well as smart chunking and caching to achieve large speedups. If installed, must be Version 2.4.6 or higher.
- **bottleneck**: for accelerating certain types of nan evaluations. **bottleneck** uses specialized cython routines to achieve large speedups. If installed, must be Version 1.0.0 or higher.

---

**Note:** You are highly encouraged to install these libraries, as they provide speed improvements, especially when working with large data sets.

---

### 2.5.2 Optional Dependencies

- **Cython**: Only necessary to build development version. Version 0.24 or higher.
- **SciPy**: miscellaneous statistical functions, Version 0.14.0 or higher
- **xarray**: pandas like handling for > 2 dims, needed for converting Panels to xarray objects. Version 0.7.0 or higher is recommended.
- **PyTables**: necessary for HDF5-based storage. Version 3.0.0 or higher required, Version 3.2.1 or higher highly recommended.
- **Feather Format**: necessary for feather-based storage, version 0.3.1 or higher.
- **Apache Parquet**, either **pyarrow** (>= 0.4.1) or **fastparquet** (>= 0.0.6) for parquet-based storage. The **snappy** and **brotli** are available for compression support.
- **SQLAlchemy**: for SQL database support. Version 0.8.1 or higher recommended. Besides SQLAlchemy, you also need a database specific driver. You can find an overview of supported drivers for each SQL dialect in the [SQLAlchemy docs](#). Some common drivers are:
  - **psycopg2**: for PostgreSQL

- `pymysql`: for MySQL.
- `SQLite`: for SQLite, this is included in Python’s standard library by default.
- `matplotlib`: for plotting, Version 1.4.3 or higher.
- For Excel I/O:
  - `xlrd/xlwt`: Excel reading (`xlrd`) and writing (`xlwt`)
  - `openpyxl`: `openpyxl` version 2.4.0 for writing `.xlsx` files (`xlrd`  $\geq$  0.9.0)
  - `XlsxWriter`: Alternative Excel writer
- `Jinja2`: Template engine for conditional HTML formatting.
- `s3fs`: necessary for Amazon S3 access (`s3fs`  $\geq$  0.0.7).
- `blosc`: for msgpack compression using `blosc`
- One of `qtpy` (requires `PyQt` or `PySide`), `PyQt5`, `PyQt4`, `pygtk`, `xsel`, or `xclip`: necessary to use `read_clipboard()`. Most package managers on Linux distributions will have `xclip` and/or `xsel` immediately available for installation.
- `pandas-gbq`: for Google BigQuery I/O.
- `Backports.lzma`: Only for Python 2, for writing to and/or reading from an xz compressed DataFrame in CSV; Python 3 support is built into the standard library.
- One of the following combinations of libraries is needed to use the top-level `read_html()` function:  
Changed in version 0.23.0.

---

**Note:** If using BeautifulSoup4 a minimum version of 4.2.1 is required

---

- `BeautifulSoup4` and `html5lib` (Any recent version of `html5lib` is okay.)
- `BeautifulSoup4` and `lxml`
- `BeautifulSoup4` and `html5lib` and `lxml`
- Only `lxml`, although see *HTML Table Parsing* for reasons as to why you should probably **not** take this approach.

**Warning:**

- if you install `BeautifulSoup4` you must install either `lxml` or `html5lib` or both. `read_html()` will **not** work with *only* `BeautifulSoup4` installed.
- You are highly encouraged to read *HTML Table Parsing gotchas*. It explains issues surrounding the installation and usage of the above three libraries.

---

**Note:**

- if you’re on a system with `apt-get` you can do

```
sudo apt-get build-dep python-lxml
```

to get the necessary dependencies for installation of `lxml`. This will prevent further headaches down the line.

---

---

**Note:** Without the optional dependencies, many useful features will not work. Hence, it is highly recommended that you install these. A packaged distribution like [Anaconda](#), [ActivePython](#) (version 2.7 or 3.5), or [Enthought Canopy](#) may be worth considering.

---



## CONTRIBUTING TO PANDAS

### Table of contents:

- *Where to start?*
- *Bug reports and enhancement requests*
- *Working with the code*
  - *Version control, Git, and GitHub*
  - *Getting started with Git*
  - *Forking*
  - *Creating a development environment*
    - \* *Installing a C Compiler*
    - \* *Creating a Python Environment*
    - \* *Creating a Python Environment (pip)*
  - *Creating a branch*
- *Contributing to the documentation*
  - *About the pandas documentation*
  - *How to build the pandas documentation*
    - \* *Requirements*
    - \* *Building the documentation*
    - \* *Building master branch documentation*
- *Contributing to the code base*
  - *Code standards*
    - \* *C (cpplint)*
    - \* *Python (PEP8)*
    - \* *Backwards Compatibility*
  - *Testing With Continuous Integration*
  - *Test-driven development/code writing*
    - \* *Writing tests*

- \* *Transitioning to pytest*
- \* *Using pytest*
  - *Running the test suite*
  - *Running the performance test suite*
  - *Documenting your code*
- *Contributing your changes to pandas*
  - *Committing your code*
  - *Pushing your changes*
  - *Review your code*
  - *Finally, make the pull request*
  - *Updating your pull request*
  - *Delete your merged branch (optional)*

## 3.1 Where to start?

All contributions, bug reports, bug fixes, documentation improvements, enhancements, and ideas are welcome.

If you are brand new to pandas or open-source development, we recommend going through the [GitHub “issues” tab](#) to find issues that interest you. There are a number of issues listed under [Docs](#) and [good first issue](#) where you could start out. Once you’ve found an interesting issue, you can return here to get your development environment setup.

Feel free to ask questions on the [mailing list](#) or on [Gitter](#).

## 3.2 Bug reports and enhancement requests

Bug reports are an important part of making *pandas* more stable. Having a complete bug report will allow others to reproduce the bug and provide insight into fixing. See [this stackoverflow article](#) and [this blogpost](#) for tips on writing a good bug report.

Trying the bug-producing code out on the *master* branch is often a worthwhile exercise to confirm the bug still exists. It is also worth searching existing bug reports and pull requests to see if the issue has already been reported and/or fixed.

Bug reports must:

1. Include a short, self-contained Python snippet reproducing the problem. You can format the code nicely by using [GitHub Flavored Markdown](#):

```
```python
>>> from pandas import DataFrame
>>> df = DataFrame(...)
...
```
```

2. Include the full version string of *pandas* and its dependencies. You can use the built in function:

```
>>> import pandas as pd
>>> pd.show_versions()
```

3. Explain why the current behavior is wrong/not desired and what you expect instead.

The issue will then show up to the *pandas* community and be open to comments/ideas from others.

## 3.3 Working with the code

Now that you have an issue you want to fix, enhancement to add, or documentation to improve, you need to learn how to work with GitHub and the *pandas* code base.

### 3.3.1 Version control, Git, and GitHub

To the new user, working with Git is one of the more daunting aspects of contributing to *pandas*. It can very quickly become overwhelming, but sticking to the guidelines below will help keep the process straightforward and mostly trouble free. As always, if you are having difficulties please feel free to ask for help.

The code is hosted on [GitHub](#). To contribute you will need to sign up for a [free GitHub account](#). We use [Git](#) for version control to allow many people to work together on the project.

Some great resources for learning Git:

- the [GitHub help pages](#).
- the [NumPy's documentation](#).
- Matthew Brett's [Pydagogue](#).

### 3.3.2 Getting started with Git

[GitHub has instructions](#) for installing git, setting up your SSH key, and configuring git. All these steps need to be completed before you can work seamlessly between your local repository and GitHub.

### 3.3.3 Forking

You will need your own fork to work on the code. Go to the [pandas project page](#) and hit the `Fork` button. You will want to clone your fork to your machine:

```
git clone https://github.com/your-user-name/pandas.git pandas-yourname
cd pandas-yourname
git remote add upstream https://github.com/pandas-dev/pandas.git
```

This creates the directory *pandas-yourname* and connects your repository to the upstream (main project) *pandas* repository.

### 3.3.4 Creating a development environment

To test out code changes, you'll need to build *pandas* from source, which requires a C compiler and Python environment. If you're making documentation changes, you can skip to [Contributing to the documentation](#) but you won't be able to build the documentation locally before pushing your changes.

### 3.3.4.1 Installing a C Compiler

Pandas uses C extensions (mostly written using Cython) to speed up certain operations. To install pandas from source, you need to compile these C extensions, which means you need a C compiler. This process depends on which platform you're using. Follow the [Cython contributing guidelines](#) for getting a compiler installed. You don't need to do any of the `./configure` or `make` steps; you only need to install the compiler.

For Windows developers, the following links may be helpful.

- <https://blogs.msdn.microsoft.com/pythonengineering/2016/04/11/unable-to-find-vcvarsall-bat/>
- <https://github.com/conda/conda-recipes/wiki/Building-from-Source-on-Windows-32-bit-and-64-bit>
- <https://cowboyprogrammer.org/building-python-wheels-for-windows/>
- <https://blog.ionelmc.ro/2014/12/21/compiling-python-extensions-on-windows/>
- <https://support.enthought.com/hc/en-us/articles/204469260-Building-Python-extensions-with-Canopy>

Let us know if you have any difficulties by opening an issue or reaching out on [Gitter](#).

### 3.3.4.2 Creating a Python Environment

Now that you have a C compiler, create an isolated pandas development environment:

- Install either [Anaconda](#) or [miniconda](#)
- Make sure your conda is up to date (`conda update conda`)
- Make sure that you have [cloned the repository](#)
- `cd` to the *pandas* source directory

We'll now kick off a three-step process:

1. Install the build dependencies
2. Build and install pandas
3. Install the optional dependencies

```
# Create and activate the build environment
conda env create -f ci/environment-dev.yaml
conda activate pandas-dev

# or with older versions of Anaconda:
source activate pandas-dev

# Build and install pandas
python setup.py build_ext --inplace -j 4
python -m pip install -e .

# Install the rest of the optional dependencies
conda install -c defaults -c conda-forge --file=ci/requirements-optional-conda.txt
```

At this point you should be able to import pandas from your locally built version:

```
$ python # start an interpreter
>>> import pandas
>>> print(pandas.__version__)
0.22.0.dev0+29.g4ad6d4d74
```

This will create the new environment, and not touch any of your existing environments, nor any existing Python installation.

To view your environments:

```
conda info -e
```

To return to your root environment:

```
conda deactivate
```

See the full conda docs [here](#).

### 3.3.4.3 Creating a Python Environment (pip)

If you aren't using conda for your development environment, follow these instructions. You'll need to have at least python3.5 installed on your system.

```
# Create a virtual environment
# Use an ENV_DIR of your choice. We'll use ~/virtualenvs/pandas-dev
# Any parent directories should already exist
python3 -m venv ~/virtualenvs/pandas-dev
# Activate the virtualenv
. ~/virtualenvs/pandas-dev/bin/activate

# Install the build dependencies
python -m pip install -r ci/requirements_dev.txt
# Build and install pandas
python setup.py build_ext --inplace -j 4
python -m pip install -e .

# Install additional dependencies
python -m pip install -r ci/requirements-optional-pip.txt
```

### 3.3.5 Creating a branch

You want your master branch to reflect only production-ready code, so create a feature branch for making your changes. For example:

```
git branch shiny-new-feature
git checkout shiny-new-feature
```

The above can be simplified to:

```
git checkout -b shiny-new-feature
```

This changes your working directory to the shiny-new-feature branch. Keep any changes in this branch specific to one bug or feature so it is clear what the branch brings to *pandas*. You can have many shiny-new-features and switch in between them using the git checkout command.

When creating this branch, make sure your master branch is up to date with the latest upstream master version. To update your local master branch, you can do:

```
git checkout master
git pull upstream master --ff-only
```

When you want to update the feature branch with changes in master after you created the branch, check the section on *updating a PR*.

## 3.4 Contributing to the documentation

Contributing to the documentation benefits everyone who uses *pandas*. We encourage you to help us improve the documentation, and you don't have to be an expert on *pandas* to do so! In fact, there are sections of the docs that are worse off after being written by experts. If something in the docs doesn't make sense to you, updating the relevant section after you figure it out is a great way to ensure it will help the next person.

### Documentation:

- *About the pandas documentation*
- *How to build the pandas documentation*
  - *Requirements*
  - *Building the documentation*
  - *Building master branch documentation*

### 3.4.1 About the *pandas* documentation

The documentation is written in **reStructuredText**, which is almost like writing in plain English, and built using *Sphinx*. The Sphinx Documentation has an excellent [introduction to reST](#). Review the Sphinx docs to perform more complex changes to the documentation as well.

Some other important things to know about the docs:

- The *pandas* documentation consists of two parts: the docstrings in the code itself and the docs in this folder `pandas/doc/`.

The docstrings provide a clear explanation of the usage of the individual functions, while the documentation in this folder consists of tutorial-like overviews per topic together with some other information (what's new, installation, etc).

- The docstrings follow a pandas convention, based on the **Numpy Docstring Standard**. Follow the *pandas docstring guide* for detailed instructions on how to write a correct docstring.

#### 3.4.1.1 pandas docstring guide

---

**Note:** [Video tutorial: Pandas docstring guide](#) by Frank Akogun.

---

### About docstrings and standards

A Python docstring is a string used to document a Python module, class, function or method, so programmers can understand what it does without having to read the details of the implementation.

Also, it is a common practice to generate online (html) documentation automatically from docstrings. *Sphinx* serves this purpose.

Next example gives an idea on how a docstring looks like:

```
def add(num1, num2):
    """
    Add up two integer numbers.

    This function simply wraps the `+` operator, and does not
    do anything interesting, except for illustrating what is
    the docstring of a very simple function.

    Parameters
    -----
    num1 : int
        First number to add
    num2 : int
        Second number to add

    Returns
    -----
    int
        The sum of `num1` and `num2`

    See Also
    -----
    subtract : Subtract one integer from another

    Examples
    -----
    >>> add(2, 2)
    4
    >>> add(25, 0)
    25
    >>> add(10, -10)
    0
    """
    return num1 + num2
```

Some standards exist about docstrings, so they are easier to read, and they can be exported to other formats such as html or pdf.

The first conventions every Python docstring should follow are defined in [PEP-257](#).

As PEP-257 is quite open, and some other standards exist on top of it. In the case of pandas, the numpy docstring convention is followed. The conventions is explained in this document:

- [numpydoc docstring guide](#) (which is based in the original [Guide to NumPy/SciPy documentation](#))

numpydoc is a Sphinx extension to support the numpy docstring convention.

The standard uses reStructuredText (reST). reStructuredText is a markup language that allows encoding styles in plain text files. Documentation about reStructuredText can be found in:

- [Sphinx reStructuredText primer](#)
- [Quick reStructuredText reference](#)
- [Full reStructuredText specification](#)

Pandas has some helpers for sharing docstrings between related classes, see [Sharing Docstrings](#).

The rest of this document will summarize all the above guides, and will provide additional convention specific to the pandas project.

## Writing a docstring

### General rules

Docstrings must be defined with three double-quotes. No blank lines should be left before or after the docstring. The text starts in the next line after the opening quotes. The closing quotes have their own line (meaning that they are not at the end of the last sentence).

In rare occasions reST styles like bold text or italics will be used in docstrings, but it is common to have inline code, which is presented between backticks. It is considered inline code:

- The name of a parameter
- Python code, a module, function, built-in, type, literal... (e.g. `os`, `list`, `numpy.abs`, `datetime.date`, `True`)
- A pandas class (in the form `:class:`pandas.Series``)
- A pandas method (in the form `:meth:`pandas.Series.sum``)
- A pandas function (in the form `:func:`pandas.to_datetime``)

---

**Note:** To display only the last component of the linked class, method or function, prefix it with `~`. For example, `:class:`~pandas.Series`` will link to `pandas.Series` but only display the last part, `Series` as the link text. See [Sphinx cross-referencing syntax](#) for details.

---

#### Good:

```
def add_values(arr):  
    """  
    Add the values in `arr`.  
  
    This is equivalent to Python `sum` of :meth:`pandas.Series.sum`.  
  
    Some sections are omitted here for simplicity.  
    """  
    return sum(arr)
```

#### Bad:

```
def func():  
  
    """Some function.  
  
    With several mistakes in the docstring.  
  
    It has a blank line after the signature `def func():`.  
  
    The text 'Some function' should go in the line after the  
    opening quotes of the docstring, not in the same line.  
  
    There is a blank line between the docstring and the first line  
    of code `foo = 1`.  
  
    The closing quotes should be in the next line, not in this one."""  
  
    foo = 1
```

(continues on next page)



(continued from previous page)

```
bar = 2
return foo + bar
```

## Section 1: Short summary

The short summary is a single sentence that expresses what the function does in a concise way.

The short summary must start with a capital letter, end with a dot, and fit in a single line. It needs to express what the object does without providing details. For functions and methods, the short summary must start with an infinitive verb.

### Good:

```
def astype(dtype):
    """
    Cast Series type.

    This section will provide further details.
    """
    pass
```

### Bad:

```
def astype(dtype):
    """
    Casts Series type.

    Verb in third-person of the present simple, should be infinitive.
    """
    pass

def astype(dtype):
    """
    Method to cast Series type.

    Does not start with verb.
    """
    pass

def astype(dtype):
    """
    Cast Series type

    Missing dot at the end.
    """
    pass

def astype(dtype):
    """
    Cast Series type from its current type to the new type defined in
    the parameter dtype.

    Summary is too verbose and doesn't fit in a single line.
    """
    pass
```

## Section 2: Extended summary

The extended summary provides details on what the function does. It should not go into the details of the parameters, or discuss implementation notes, which go in other sections.

A blank line is left between the short summary and the extended summary. And every paragraph in the extended summary is finished by a dot.

The extended summary should provide details on why the function is useful and their use cases, if it is not too generic.

```
def unstack():
    """
    Pivot a row index to columns.

    When using a multi-index, a level can be pivoted so each value in
    the index becomes a column. This is especially useful when a subindex
    is repeated for the main index, and data is easier to visualize as a
    pivot table.

    The index level will be automatically removed from the index when added
    as columns.
    """
    pass
```

## Section 3: Parameters

The details of the parameters will be added in this section. This section has the title “Parameters”, followed by a line with a hyphen under each letter of the word “Parameters”. A blank line is left before the section title, but not after, and not between the line with the word “Parameters” and the one with the hyphens.

After the title, each parameter in the signature must be documented, including *\*args* and *\*\*kwargs*, but not *self*.

The parameters are defined by their name, followed by a space, a colon, another space, and the type (or types). Note that the space between the name and the colon is important. Types are not defined for *\*args* and *\*\*kwargs*, but must be defined for all other parameters. After the parameter definition, it is required to have a line with the parameter description, which is indented, and can have multiple lines. The description must start with a capital letter, and finish with a dot.

For keyword arguments with a default value, the default will be listed after a comma at the end of the type. The exact form of the type in this case will be “int, default 0”. In some cases it may be useful to explain what the default argument means, which can be added after a comma “int, default -1, meaning all cpus”.

In cases where the default value is *None*, meaning that the value will not be used. Instead of “str, default None”, it is preferred to write “str, optional”. When *None* is a value being used, we will keep the form “str, default None”. For example, in *df.to\_csv(compression=None)*, *None* is not a value being used, but means that compression is optional, and no compression is being used if not provided. In this case we will use *str, optional*. Only in cases like *func(value=None)* and *None* is being used in the same way as *0* or *foo* would be used, then we will specify “str, int or None, default None”.

**Good:**

```
class Series:
    def plot(self, kind, color='blue', **kwargs):
        """
        Generate a plot.
```

(continues on next page)

(continued from previous page)

```

Render the data in the Series as a matplotlib plot of the
specified kind.

Parameters
-----
kind : str
    Kind of matplotlib plot.
color : str, default 'blue'
    Color name or rgb code.
**kwargs
    These parameters will be passed to the matplotlib plotting
    function.
"""
pass

```

**Bad:**

```

class Series:
    def plot(self, kind, **kwargs):
        """
        Generate a plot.

        Render the data in the Series as a matplotlib plot of the
        specified kind.

        Note the blank line between the parameters title and the first
        parameter. Also, note that after the name of the parameter `kind`
        and before the colon, a space is missing.

        Also, note that the parameter descriptions do not start with a
        capital letter, and do not finish with a dot.

        Finally, the `**kwargs` parameter is missing.

        Parameters
        -----

        kind: str
            kind of matplotlib plot
        """
        pass

```

**Parameter types**

When specifying the parameter types, Python built-in data types can be used directly (the Python type is preferred to the more verbose string, integer, boolean, etc):

- int
- float
- str
- bool

For complex types, define the subtypes. For *dict* and *tuple*, as more than one type is present, we use the brackets to help read the type (curly brackets for *dict* and normal brackets for *tuple*):

- list of int
- dict of {str : int}
- tuple of (str, int, int)
- tuple of (str,)
- set of str

In case where there are just a set of values allowed, list them in curly brackets and separated by commas (followed by a space). If the values are ordinal and they have an order, list them in this order. Otherwise, list the default value first, if there is one:

- {0, 10, 25}
- {'simple', 'advanced'}
- {'low', 'medium', 'high'}
- {'cat', 'dog', 'bird'}

If the type is defined in a Python module, the module must be specified:

- datetime.date
- datetime.datetime
- decimal.Decimal

If the type is in a package, the module must be also specified:

- numpy.ndarray
- scipy.sparse.coo\_matrix

If the type is a pandas type, also specify pandas except for Series and DataFrame:

- Series
- DataFrame
- pandas.Index
- pandas.Categorical
- pandas.SparseArray

If the exact type is not relevant, but must be compatible with a numpy array, array-like can be specified. If Any type that can be iterated is accepted, iterable can be used:

- array-like
- iterable

If more than one type is accepted, separate them by commas, except the last two types, that need to be separated by the word 'or':

- int or float
- float, decimal.Decimal or None
- str or list of str

If None is one of the accepted values, it always needs to be the last in the list.

For axis, the convention is to use something like:

- axis : {0 or 'index', 1 or 'columns', None}, default None

## Section 4: Returns or Yields

If the method returns a value, it will be documented in this section. Also if the method yields its output.

The title of the section will be defined in the same way as the “Parameters”. With the names “Returns” or “Yields” followed by a line with as many hyphens as the letters in the preceding word.

The documentation of the return is also similar to the parameters. But in this case, no name will be provided, unless the method returns or yields more than one value (a tuple of values).

The types for “Returns” and “Yields” are the same as the ones for the “Parameters”. Also, the description must finish with a dot.

For example, with a single value:

```
def sample():
    """
    Generate and return a random number.

    The value is sampled from a continuous uniform distribution between
    0 and 1.

    Returns
    -----
    float
        Random number generated.
    """
    return random.random()
```

With more than one value:

```
def random_letters():
    """
    Generate and return a sequence of random letters.

    The length of the returned string is also random, and is also
    returned.

    Returns
    -----
    length : int
        Length of the returned string.
    letters : str
        String of random letters.
    """
    length = random.randint(1, 10)
    letters = ''.join(random.choice(string.ascii_lowercase)
                      for i in range(length))
    return length, letters
```

If the method yields its value:

```
def sample_values():
    """
    Generate an infinite sequence of random numbers.

    The values are sampled from a continuous uniform distribution between
    0 and 1.
```

(continues on next page)

(continued from previous page)

```

Yields
-----
float
    Random number generated.
"""
while True:
    yield random.random()

```

## Section 5: See Also

This section is used to let users know about pandas functionality related to the one being documented. In rare cases, if no related methods or functions can be found at all, this section can be skipped.

An obvious example would be the *head()* and *tail()* methods. As *tail()* does the equivalent as *head()* but at the end of the *Series* or *DataFrame* instead of at the beginning, it is good to let the users know about it.

To give an intuition on what can be considered related, here there are some examples:

- `loc` and `iloc`, as they do the same, but in one case providing indices and in the other positions
- `max` and `min`, as they do the opposite
- `iterrows`, `itertuples` and `iteritems`, as it is easy that a user looking for the method to iterate over columns ends up in the method to iterate over rows, and vice-versa
- `fillna` and `dropna`, as both methods are used to handle missing values
- `read_csv` and `to_csv`, as they are complementary
- `merge` and `join`, as one is a generalization of the other
- `astype` and `pandas.to_datetime`, as users may be reading the documentation of `astype` to know how to cast as a date, and the way to do it is with `pandas.to_datetime`
- `where` is related to `numpy.where`, as its functionality is based on it

When deciding what is related, you should mainly use your common sense and think about what can be useful for the users reading the documentation, especially the less experienced ones.

When relating to other libraries (mainly `numpy`), use the name of the module first (not an alias like `np`). If the function is in a module which is not the main one, like `scipy.sparse`, list the full module (e.g. `scipy.sparse.coo_matrix`).

This section, as the previous, also has a header, “See Also” (note the capital S and A). Also followed by the line with hyphens, and preceded by a blank line.

After the header, we will add a line for each related method or function, followed by a space, a colon, another space, and a short description that illustrated what this method or function does, why is it relevant in this context, and what are the key differences between the documented function and the one referencing. The description must also finish with a dot.

Note that in “Returns” and “Yields”, the description is located in the following line than the type. But in this section it is located in the same line, with a colon in between. If the description does not fit in the same line, it can continue in the next ones, but it has to be indented in them.

For example:

```

class Series:
    def head(self):
        """
        Return the first 5 elements of the Series.

        This function is mainly useful to preview the values of the
        Series without displaying the whole of it.

        Returns
        -----
        Series
            Subset of the original series with the 5 first values.

        See Also
        -----
        Series.tail : Return the last 5 elements of the Series.
        Series.iloc : Return a slice of the elements in the Series,
                      which can also be used to return the first or last n.
        """
        return self.iloc[:5]

```

## Section 6: Notes

This is an optional section used for notes about the implementation of the algorithm. Or to document technical aspects of the function behavior.

Feel free to skip it, unless you are familiar with the implementation of the algorithm, or you discover some counter-intuitive behavior while writing the examples for the function.

This section follows the same format as the extended summary section.

## Section 7: Examples

This is one of the most important sections of a docstring, even if it is placed in the last position. As often, people understand concepts better with examples, than with accurate explanations.

Examples in docstrings, besides illustrating the usage of the function or method, must be valid Python code, that in a deterministic way returns the presented output, and that can be copied and run by users.

They are presented as a session in the Python terminal. `>>>` is used to present code. `...` is used for code continuing from the previous line. Output is presented immediately after the last line of code generating the output (no blank lines in between). Comments describing the examples can be added with blank lines before and after them.

The way to present examples is as follows:

1. Import required libraries (except `numpy` and `pandas`)
2. Create the data required for the example
3. Show a very basic example that gives an idea of the most common use case
4. Add examples with explanations that illustrate how the parameters can be used for extended functionality

A simple example could be:

```
class Series:
    def head(self, n=5):
        """
        Return the first elements of the Series.

        This function is mainly useful to preview the values of the
        Series without displaying the whole of it.

        Parameters
        -----
        n : int
            Number of values to return.

        Return
        -----
        pandas.Series
            Subset of the original series with the n first values.

        See Also
        -----
        tail : Return the last n elements of the Series.

        Examples
        -----
        >>> s = pd.Series(['Ant', 'Bear', 'Cow', 'Dog', 'Falcon',
        ...               'Lion', 'Monkey', 'Rabbit', 'Zebra'])
        >>> s.head()
        0    Ant
        1    Bear
        2    Cow
        3    Dog
        4    Falcon
        dtype: object

        With the `n` parameter, we can change the number of returned rows:

        >>> s.head(n=3)
        0    Ant
        1    Bear
        2    Cow
        dtype: object
        """
        return self.iloc[:n]
```

The examples should be as concise as possible. In cases where the complexity of the function requires long examples, is recommended to use blocks with headers in bold. Use double star **\*\*** to make a text bold, like in **\*\*this example\*\***.

### Conventions for the examples

Code in examples is assumed to always start with these two lines which are not shown:

```
import numpy as np
import pandas as pd
```

Any other module used in the examples must be explicitly imported, one per line (as recommended in [PEP-8](#))



and avoiding aliases. Avoid excessive imports, but if needed, imports from the standard library go first, followed by third-party libraries (like matplotlib).

When illustrating examples with a single `Series` use the name `s`, and if illustrating with a single `DataFrame` use the name `df`. For indices, `idx` is the preferred name. If a set of homogeneous `Series` or `DataFrame` is used, name them `s1`, `s2`, `s3...` or `df1`, `df2`, `df3...`. If the data is not homogeneous, and more than one structure is needed, name them with something meaningful, for example `df_main` and `df_to_join`.

Data used in the example should be as compact as possible. The number of rows is recommended to be around 4, but make it a number that makes sense for the specific example. For example in the `head` method, it requires to be higher than 5, to show the example with the default values. If doing the `mean`, we could use something like `[1, 2, 3]`, so it is easy to see that the value returned is the mean.

For more complex examples (grouping for example), avoid using data without interpretation, like a matrix of random numbers with columns A, B, C, D... And instead use a meaningful example, which makes it easier to understand the concept. Unless required by the example, use names of animals, to keep examples consistent. And numerical properties of them.

When calling the method, keywords arguments `head(n=3)` are preferred to positional arguments `head(3)`.

**Good:**

```
class Series:
    def mean(self):
        """
        Compute the mean of the input.

        Examples
        -----
        >>> s = pd.Series([1, 2, 3])
        >>> s.mean()
        2
        """
        pass

    def fillna(self, value):
        """
        Replace missing values by `value`.

        Examples
        -----
        >>> s = pd.Series([1, np.nan, 3])
        >>> s.fillna(0)
        [1, 0, 3]
        """
        pass

    def groupby_mean(self):
        """
        Group by index and return mean.

        Examples
        -----
        >>> s = pd.Series([380., 370., 24., 26],
        ...               name='max_speed',
        ...               index=['falcon', 'falcon', 'parrot', 'parrot'])
        >>> s.groupby_mean()
        index
```

(continues on next page)

(continued from previous page)

```

    falcon      375.0
    parrot      25.0
    Name: max_speed, dtype: float64
    """
    pass

def contains(self, pattern, case_sensitive=True, na=numpy.nan):
    """
    Return whether each value contains `pattern`.

    In this case, we are illustrating how to use sections, even
    if the example is simple enough and does not require them.

    Examples
    -----
    >>> s = pd.Series('Antelope', 'Lion', 'Zebra', numpy.nan)
    >>> s.contains(pattern='a')
    0    False
    1    False
    2     True
    3     NaN
    dtype: bool

    **Case sensitivity**

    With `case_sensitive` set to `False` we can match `a` with both
    `a` and `A`:

    >>> s.contains(pattern='a', case_sensitive=False)
    0     True
    1    False
    2     True
    3     NaN
    dtype: bool

    **Missing values**

    We can fill missing values in the output using the `na` parameter:

    >>> s.contains(pattern='a', na=False)
    0    False
    1    False
    2     True
    3    False
    dtype: bool
    """
    pass

```

**Bad:**

```

def method(foo=None, bar=None):
    """
    A sample DataFrame method.

    Do not import numpy and pandas.

    Try to use meaningful data, when it makes the example easier

```

(continues on next page)

(continued from previous page)

to understand.

Try to avoid positional arguments like in `df.method(1)`. They can be all right if previously defined with a meaningful name, like in `present_value(interest_rate)`, but avoid them otherwise.

When presenting the behavior with different parameters, do not place all the calls one next to the other. Instead, add a short sentence explaining what the example shows.

Examples

-----

```
>>> import numpy as np
>>> import pandas as pd
>>> df = pd.DataFrame(numpy.random.randn(3, 3),
...                    columns=('a', 'b', 'c'))
>>> df.method(1)
21
>>> df.method(bar=14)
123
"""
pass
```

### Tips for getting your examples pass the doctests

Getting the examples pass the doctests in the validation script can sometimes be tricky. Here are some attention points:

- Import all needed libraries (except for pandas and numpy, those are already imported as `import pandas as pd` and `import numpy as np`) and define all variables you use in the example.
- Try to avoid using random data. However random data might be OK in some cases, like if the function you are documenting deals with probability distributions, or if the amount of data needed to make the function result meaningful is too much, such that creating it manually is very cumbersome. In those cases, always use a fixed random seed to make the generated examples predictable. Example:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame({'normal': np.random.normal(100, 5, 20)})
```

- If you have a code snippet that wraps multiple lines, you need to use `'...'` on the continued lines:

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], index=['a', 'b', 'c'],
...                    columns=['A', 'B'])
```

- If you want to show a case where an exception is raised, you can do:

```
>>> pd.to_datetime(["712-01-01"])
Traceback (most recent call last):
OutOfBoundsDatetime: Out of bounds nanosecond timestamp: 712-01-01 00:00:00
```

It is essential to include the “Traceback (most recent call last):”, but for the actual error only the error name is sufficient.

- If there is a small part of the result that can vary (e.g. a hash in an object representation), you can use `...` to represent this part.

If you want to show that `s.plot()` returns a matplotlib `AxesSubplot` object, this will fail the doctest

```
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at 0x7efd0c0b0690>
```

However, you can do (notice the comment that needs to be added)

```
>>> s.plot()
<matplotlib.axes._subplots.AxesSubplot at ...>
```

## Plots in examples

There are some methods in pandas returning plots. To render the plots generated by the examples in the documentation, the `.. plot::` directive exists.

To use it, place the next code after the “Examples” header as shown below. The plot will be generated automatically when building the documentation.

```
class Series:
    def plot(self):
        """
        Generate a plot with the `Series` data.

        Examples
        -----

        .. plot::
            :context: close-figs

            >>> s = pd.Series([1, 2, 3])
            >>> s.plot()

        """
    pass
```

## Sharing Docstrings

Pandas has a system for sharing docstrings, with slight variations, between classes. This helps us keep docstrings consistent, while keeping things clear for the user reading. It comes at the cost of some complexity when writing.

Each shared docstring will have a base template with variables, like `%(klass)s`. The variables filled in later on using the `Substitution` decorator. Finally, docstrings can be appended to with the `Appender` decorator.

In this example, we’ll create a parent docstring normally (this is like `pandas.core.generic.NDFrame`). Then we’ll have two children (like `pandas.core.series.Series` and `pandas.core.frame.DataFrame`). We’ll substitute the children’s class names in this docstring.

```
class Parent:
    def my_function(self):
        """Apply my function to %(klass)s."""
        ...

class ChildA(Parent):
    @Substitution(klass="ChildA")
    @Appender(Parent.my_function.__doc__)
    def my_function(self):
```

(continues on next page)

(continued from previous page)

```

...

class ChildB(Parent):
    @Substitution(klass="ChildB")
    @Appender(Parent.my_function.__doc__)
    def my_function(self):
        ...

```

The resulting docstrings are

```

>>> print(Parent.my_function.__doc__)
Apply my function to %(klass)s.
>>> print(ChildA.my_function.__doc__)
Apply my function to ChildA.
>>> print(ChildB.my_function.__doc__)
Apply my function to ChildB.

```

Notice two things:

1. We “append” the parent docstring to the children docstrings, which are initially empty.
2. Python decorators are applied inside out. So the order is Append then Substitution, even though Substitution comes first in the file.

Our files will often contain a module-level `_shared_doc_kwargs` with some common substitution values (things like `klass`, `axes`, etc).

You can substitute and append in one shot with something like

```

@Appender(template % _shared_doc_kwargs)
def my_function(self):
    ...

```

where `template` may come from a module-level `_shared_docs` dictionary mapping function names to docstrings. Wherever possible, we prefer using `Appender` and `Substitution`, since the docstring-writing processes is slightly closer to normal.

See `pandas.core.generic.NDFrame.fillna` for an example `template`, and `pandas.core.series.Series.fillna` and `pandas.core.generic.frame.fillna` for the filled versions.

- The tutorials make heavy use of the `ipython directive` sphinx extension. This directive lets you put code in the documentation which will be run during the doc build. For example:

```

.. ipython:: python

    x = 2
    x**3

```

will be rendered as:

```

In [1]: x = 2

In [2]: x**3
Out[2]: 8

```

Almost all code examples in the docs are run (and the output saved) during the doc build. This approach means that code examples will always be up to date, but it does make the doc building a bit more complex.

- Our API documentation in `doc/source/api.rst` houses the auto-generated documentation from the docstrings. For classes, there are a few subtleties around controlling which methods and attributes have pages auto-generated.

We have two autosummary templates for classes.

1. `_templates/autosummary/class.rst`. Use this when you want to automatically generate a page for every public method and attribute on the class. The `Attributes` and `Methods` sections will be automatically added to the class' rendered documentation by `numpydoc`. See `DataFrame` for an example.
2. `_templates/autosummary/class_without_autosummary`. Use this when you want to pick a subset of methods / attributes to auto-generate pages for. When using this template, you should include an `Attributes` and `Methods` section in the class docstring. See `CategoricalIndex` for an example.

Every method should be included in a `toctree` in `api.rst`, else Sphinx will emit a warning.

---

**Note:** The `.rst` files are used to automatically generate Markdown and HTML versions of the docs. For this reason, please do not edit `CONTRIBUTING.md` directly, but instead make any changes to `doc/source/contributing.rst`. Then, to generate `CONTRIBUTING.md`, use `pandoc` with the following command:

```
pandoc doc/source/contributing.rst -t markdown_github > CONTRIBUTING.md
```

The utility script `scripts/validate_docstrings.py` can be used to get a csv summary of the API documentation. And also validate common errors in the docstring of a specific class, function or method. The summary also compares the list of methods documented in `doc/source/api.rst` (which is used to generate the [API Reference](#) page) and the actual public methods. This will identify methods documented in `doc/source/api.rst` that are not actually class methods, and existing methods that are not documented in `doc/source/api.rst`.

## 3.4.2 How to build the *pandas* documentation

### 3.4.2.1 Requirements

First, you need to have a development environment to be able to build pandas (see the docs on [creating a development environment above](#)).

### 3.4.2.2 Building the documentation

So how do you build the docs? Navigate to your local `pandas/doc/` directory in the console and run:

```
python make.py html
```

Then you can find the HTML output in the folder `pandas/doc/build/html/`.

The first time you build the docs, it will take quite a while because it has to run all the code examples and build all the generated docstring pages. In subsequent evocations, sphinx will try to only build the pages that have been modified.

If you want to do a full clean build, do:

```
python make.py clean
python make.py html
```

You can tell `make.py` to compile only a single section of the docs, greatly reducing the turn-around time for checking your changes.

```
# omit autosummary and API section
python make.py clean
python make.py --no-api

# compile the docs with only a single
# section, that which is in indexing.rst
python make.py clean
python make.py --single indexing

# compile the reference docs for a single function
python make.py clean
python make.py --single DataFrame.join
```

For comparison, a full documentation build may take 15 minutes, but a single section may take 15 seconds. Subsequent builds, which only process portions you have changed, will be faster.

You can also specify to use multiple cores to speed up the documentation build:

```
python make.py html --num-jobs 4
```

Open the following file in a web browser to see the full documentation you just built:

```
pandas/docs/build/html/index.html
```

And you'll have the satisfaction of seeing your new and improved documentation!

### 3.4.2.3 Building master branch documentation

When pull requests are merged into the *pandas* master branch, the main parts of the documentation are also built by Travis-CI. These docs are then hosted [here](#), see also the *Continuous Integration* section.

## 3.5 Contributing to the code base

### Code Base:

- *Code standards*
  - *C (cpplint)*
  - *Python (PEP8)*
  - *Backwards Compatibility*
- *Testing With Continuous Integration*
- *Test-driven development/code writing*
  - *Writing tests*
  - *Transitioning to pytest*
  - *Using pytest*
- *Running the test suite*
- *Running the performance test suite*

- *Documenting your code*

### 3.5.1 Code standards

Writing good code is not just about what you write. It is also about *how* you write it. During *Continuous Integration* testing, several tools will be run to check your code for stylistic errors. Generating any warnings will cause the test to fail. Thus, good style is a requirement for submitting code to *pandas*.

In addition, because a lot of people use our library, it is important that we do not make sudden changes to the code that could have the potential to break a lot of user code as a result, that is, we need it to be as *backwards compatible* as possible to avoid mass breakages.

Additional standards are outlined on the [code style wiki page](#).

#### 3.5.1.1 C (cpplint)

*pandas* uses the [Google](#) standard. Google provides an open source style checker called `cpplint`, but we use a fork of it that can be found [here](#). Here are *some* of the more common `cpplint` issues:

- we restrict line-length to 80 characters to promote readability
- every header file must include a header guard to avoid name collisions if re-included

*Continuous Integration* will run the `cpplint` tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself:

```
cpplint --extensions=c,h --headers=h --filter=--readability/casting,-runtime/int,-  
↳build/include_subdir modified-c-file
```

You can also run this command on an entire directory if necessary:

```
cpplint --extensions=c,h --headers=h --filter=--readability/casting,-runtime/int,-  
↳build/include_subdir --recursive modified-c-directory
```

To make your commits compliant with this standard, you can install the [ClangFormat](#) tool, which can be downloaded [here](#). To configure, in your home directory, run the following command:

```
clang-format style=google -dump-config > .clang-format
```

Then modify the file to ensure that any indentation width parameters are at least four. Once configured, you can run the tool as follows:

```
clang-format modified-c-file
```

This will output what your file will look like if the changes are made, and to apply them, run the following command:

```
clang-format -i modified-c-file
```

To run the tool on an entire directory, you can run the following analogous commands:

```
clang-format modified-c-directory/*.c modified-c-directory/*.h  
clang-format -i modified-c-directory/*.c modified-c-directory/*.h
```

Do note that this tool is best-effort, meaning that it will try to correct as many errors as possible, but it may not correct *all* of them. Thus, it is recommended that you run `cpplint` to double check and make any other style fixes manually.



### 3.5.1.2 Python (PEP8)

*pandas* uses the [PEP8](#) standard. There are several tools to ensure you abide by this standard. Here are *some* of the more common PEP8 issues:

- we restrict line-length to 79 characters to promote readability
- passing arguments should have spaces after commas, e.g. `foo(arg1, arg2, kw1='bar')`

*Continuous Integration* will run the [flake8](#) tool and report any stylistic errors in your code. Therefore, it is helpful before submitting code to run the check yourself on the diff:

```
git diff master -u -- "*.py" | flake8 --diff
```

This command will catch any stylistic errors in your changes specifically, but be beware it may not catch all of them. For example, if you delete the only usage of an imported function, it is stylistically incorrect to import an unused function. However, style-checking the diff will not catch this because the actual import is not part of the diff. Thus, for completeness, you should run this command, though it will take longer:

```
git diff master --name-only -- "*.py" | grep "pandas/" | xargs -r flake8
```

Note that on OSX, the `-r` flag is not available, so you have to omit it and run this slightly modified command:

```
git diff master --name-only -- "*.py" | grep "pandas/" | xargs flake8
```

Note that on Windows, these commands are unfortunately not possible because commands like `grep` and `xargs` are not available natively. To imitate the behavior with the commands above, you should run:

```
git diff master --name-only -- "*.py"
```

This will list all of the Python files that have been modified. The only ones that matter during linting are any whose directory filepath begins with “pandas.” For each filepath, copy and paste it after the `flake8` command as shown below:

```
flake8 <python-filepath>
```

Alternatively, you can install the `grep` and `xargs` commands via the [MinGW](#) toolchain, and it will allow you to run the commands above.

### 3.5.1.3 Backwards Compatibility

Please try to maintain backward compatibility. *pandas* has lots of users with lots of existing code, so don’t break it if at all possible. If you think breakage is required, clearly state why as part of the pull request. Also, be careful when changing method signatures and add deprecation warnings where needed. Also, add the deprecated sphinx directive to the deprecated functions or methods.

If a function with the same arguments as the one being deprecated exist, you can use the `pandas.util._decorators.deprecate`:

```
from pandas.util._decorators import deprecate

deprecate('old_func', 'new_func', '0.21.0')
```

Otherwise, you need to do it manually:

```
def old_func():
    """Summary of the function.

    .. deprecated:: 0.21.0
       Use new_func instead.
    """
    warnings.warn('Use new_func instead.', FutureWarning, stacklevel=2)
    new_func()
```

### 3.5.2 Testing With Continuous Integration

The *pandas* test suite will run automatically on [Travis-CI](#), [Appveyor](#), and [Circle CI](#) continuous integration services, once your pull request is submitted. However, if you wish to run the test suite on a branch prior to submitting the pull request, then the continuous integration services need to be hooked to your GitHub repository. Instructions are here for [Travis-CI](#), [Appveyor](#), and [CircleCI](#).

A pull-request will be considered for merging when you have an all ‘green’ build. If any tests are failing, then you will get a red ‘X’, where you can click through to see the individual failed tests. This is an example of a green build.

Add more commits by pushing to the **period-datetime-overflow** branch on **gfyoun/pandas**.



The screenshot shows a GitHub pull request status bar with a green checkmark icon on the left. The main status is "All checks have passed" with a link to "Hide all checks". Below this, there are five individual check items, each with a green checkmark, an icon, a description, and a "Details" link:

- ci/circleci** — Your tests passed on CircleCI! [Details](#)
- codecov/patch** — 100% of diff hit (target 50%) [Details](#)
- codecov/project** — 91.03% (target 82%) [Details](#)
- continuous-integration/appveyor/pr** — AppVeyor build succeeded [Details](#)
- continuous-integration/travis-ci/pr** — The Travis CI build passed [Details](#)

At the bottom, there is a summary check: "This branch has no conflicts with the base branch" with a note: "Only those with [write access](#) to this repository can merge pull requests."

---

**Note:** Each time you push to *your* fork, a *new* run of the tests will be triggered on the CI. Appveyor will auto-cancel any non-currently-running tests for that same pull-request. You can enable the auto-cancel feature for [Travis-CI](#) here and for [CircleCI](#) here.

---

### 3.5.3 Test-driven development/code writing

*pandas* is serious about testing and strongly encourages contributors to embrace [test-driven development \(TDD\)](#). This development process “relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount

of code to pass that test.” So, before actually writing any code, you should write your tests. Often the test can be taken from the original GitHub issue. However, it is always worth considering additional use cases and writing corresponding tests.

Adding tests is one of the most common requests after code is pushed to *pandas*. Therefore, it is worth getting in the habit of writing tests ahead of time so this is never an issue.

Like many packages, *pandas* uses [pytest](#) and the convenient extensions in [numpy.testing](#).

---

**Note:** The earliest supported pytest version is 3.1.0.

---

### 3.5.3.1 Writing tests

All tests should go into the `tests` subdirectory of the specific package. This folder contains many current examples of tests, and we suggest looking to these for inspiration. If your test requires working with files or network connectivity, there is more information on the [testing page](#) of the wiki.

The `pandas.util.testing` module has many special `assert` functions that make it easier to make statements about whether Series or DataFrame objects are equivalent. The easiest way to verify that your code is correct is to explicitly construct the result you expect, then compare the actual result to the expected correct result:

```
def test_pivot(self):
    data = {
        'index' : ['A', 'B', 'C', 'C', 'B', 'A'],
        'columns' : ['One', 'One', 'One', 'Two', 'Two', 'Two'],
        'values' : [1., 2., 3., 3., 2., 1.]
    }

    frame = DataFrame(data)
    pivoted = frame.pivot(index='index', columns='columns', values='values')

    expected = DataFrame({
        'One' : {'A' : 1., 'B' : 2., 'C' : 3.},
        'Two' : {'A' : 1., 'B' : 2., 'C' : 3.}
    })

    assert_frame_equal(pivoted, expected)
```

### 3.5.3.2 Transitioning to pytest

*pandas* existing test structure is *mostly* classed based, meaning that you will typically find tests wrapped in a class.

```
class TestReallyCoolFeature(object):
    ....
```

Going forward, we are moving to a more *functional* style using the [pytest](#) framework, which offers a richer testing framework that will facilitate testing and developing. Thus, instead of writing test classes, we will write test functions like this:

```
def test_really_cool_feature():
    ....
```

### 3.5.3.3 Using pytest

Here is an example of a self-contained set of tests that illustrate multiple features that we like to use.

- functional style: tests are like `test_*` and *only* take arguments that are either fixtures or parameters
- `pytest.mark` can be used to set metadata on test functions, e.g. `skip` or `xfail`.
- using `parametrize`: allow testing of multiple cases
- to set a mark on a parameter, `pytest.param(..., marks=...)` syntax should be used
- `fixture`, code for object construction, on a per-test basis
- using bare `assert` for scalars and truth-testing
- `tm.assert_series_equal` (and its counter part `tm.assert_frame_equal`), for pandas object comparisons.
- the typical pattern of constructing an expected and comparing versus the result

We would name this file `test_cool_feature.py` and put in an appropriate place in the `pandas/tests/` structure.

```
import pytest
import numpy as np
import pandas as pd
from pandas.util import testing as tm

@pytest.mark.parametrize('dtype', ['int8', 'int16', 'int32', 'int64'])
def test_dtypes(dtype):
    assert str(np.dtype(dtype)) == dtype

@pytest.mark.parametrize('dtype', ['float32',
    pytest.param('int16', marks=pytest.mark.skip),
    pytest.param('int32',
        marks=pytest.mark.xfail(reason='to show how it works'))])
def test_mark(dtype):
    assert str(np.dtype(dtype)) == 'float32'

@pytest.fixture
def series():
    return pd.Series([1, 2, 3])

@pytest.fixture(params=['int8', 'int16', 'int32', 'int64'])
def dtype(request):
    return request.param

def test_series(series, dtype):
    result = series.astype(dtype)
    assert result.dtype == dtype

    expected = pd.Series([1, 2, 3], dtype=dtype)
    tm.assert_series_equal(result, expected)
```

A test run of this yields

```
((pandas) bash-3.2$ pytest test_cool_feature.py -v
===== test session starts =====
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.31, pluggy-0.4.0
collected 11 items
```

(continues on next page)

(continued from previous page)

```

tester.py::test_dtypes[int8] PASSED
tester.py::test_dtypes[int16] PASSED
tester.py::test_dtypes[int32] PASSED
tester.py::test_dtypes[int64] PASSED
tester.py::test_mark[float32] PASSED
tester.py::test_mark[int16] SKIPPED
tester.py::test_mark[int32] xfail
tester.py::test_series[int8] PASSED
tester.py::test_series[int16] PASSED
tester.py::test_series[int32] PASSED
tester.py::test_series[int64] PASSED

```

Tests that we have parametrized are now accessible via the test name, for example we could run these with `-k int8` to sub-select *only* those tests which match `int8`.

```

((pandas) bash-3.2$ pytest test_cool_feature.py -v -k int8
===== test session starts =====
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.31, pluggy-0.4.0
collected 11 items

test_cool_feature.py::test_dtypes[int8] PASSED
test_cool_feature.py::test_series[int8] PASSED

```

### 3.5.4 Running the test suite

The tests can then be run directly inside your Git clone (without having to install *pandas*) by typing:

```
pytest pandas
```

The tests suite is exhaustive and takes around 20 minutes to run. Often it is worth running only a subset of tests first around your changes before running the entire suite.

The easiest way to do this is with:

```
pytest pandas/path/to/test.py -k regex_matching_test_name
```

Or with one of the following constructs:

```

pytest pandas/tests/[test-module].py
pytest pandas/tests/[test-module].py::[TestClass]
pytest pandas/tests/[test-module].py::[TestClass]::[test_method]

```

Using *pytest-xdist*, one can speed up local testing on multicore machines. To use this feature, you will need to install *pytest-xdist* via:

```
pip install pytest-xdist
```

Two scripts are provided to assist with this. These scripts distribute testing across 4 threads.

On Unix variants, one can type:

```
test_fast.sh
```

On Windows, one can type:

```
test_fast.bat
```

This can significantly reduce the time it takes to locally run tests before submitting a pull request.

For more, see the [pytest](#) documentation.

New in version 0.20.0.

Furthermore one can run

```
pd.test()
```

with an imported pandas to run tests similarly.

### 3.5.5 Running the performance test suite

Performance matters and it is worth considering whether your code has introduced performance regressions. *pandas* is in the process of migrating to [asv benchmarks](#) to enable easy monitoring of the performance of critical *pandas* operations. These benchmarks are all found in the `pandas/asv_bench` directory. *asv* supports both python2 and python3.

To use all features of *asv*, you will need either `conda` or `virtualenv`. For more details please check the [asv installation webpage](#).

To install *asv*:

```
pip install git+https://github.com/spacetelescope/asv
```

If you need to run a benchmark, change your directory to `asv_bench/` and run:

```
asv continuous -f 1.1 upstream/master HEAD
```

You can replace `HEAD` with the name of the branch you are working on, and report benchmarks that changed by more than 10%. The command uses `conda` by default for creating the benchmark environments. If you want to use `virtualenv` instead, write:

```
asv continuous -f 1.1 -E virtualenv upstream/master HEAD
```

The `-E virtualenv` option should be added to all *asv* commands that run benchmarks. The default value is defined in `asv.conf.json`.

Running the full test suite can take up to one hour and use up to 3GB of RAM. Usually it is sufficient to paste only a subset of the results into the pull request to show that the committed changes do not cause unexpected performance regressions. You can run specific benchmarks using the `-b` flag, which takes a regular expression. For example, this will only run tests from a `pandas/asv_bench/benchmarks/groupby.py` file:

```
asv continuous -f 1.1 upstream/master HEAD -b ^groupby
```

If you want to only run a specific group of tests from a file, you can do it using `.` as a separator. For example:

```
asv continuous -f 1.1 upstream/master HEAD -b groupby.GroupByMethods
```

will only run the `GroupByMethods` benchmark defined in `groupby.py`.

You can also run the benchmark suite using the version of *pandas* already installed in your current Python environment. This can be useful if you do not have `virtualenv` or `conda`, or are using the `setup.py develop` approach discussed above; for the in-place build you need to set `PYTHONPATH`, e.g. `PYTHONPATH="$PWD/.."` `asv [remaining arguments]`. You can run benchmarks using an existing Python environment by:

```
asv run -e -E existing
```

or, to use a specific Python interpreter,:

```
asv run -e -E existing:python3.5
```

This will display stderr from the benchmarks, and use your local `python` that comes from your `$PATH`.

Information on how to write a benchmark and how to use `asv` can be found in the [asv documentation](#).

### 3.5.6 Documenting your code

Changes should be reflected in the release notes located in `doc/source/whatsnew/vx.y.z.txt`. This file contains an ongoing change log for each release. Add an entry to this file to document your fix, enhancement or (unavoidable) breaking change. Make sure to include the GitHub issue number when adding your entry (using `:issue:`1234`` where 1234 is the issue/pull request number).

If your code is an enhancement, it is most likely necessary to add usage examples to the existing documentation. This can be done following the section regarding documentation [above](#). Further, to let users know when this feature was added, the `versionadded` directive is used. The sphinx syntax for that is:

```
.. versionadded:: 0.21.0
```

This will put the text *New in version 0.21.0* wherever you put the sphinx directive. This should also be put in the docstring when adding a new function or method ([example](#)) or a new keyword argument ([example](#)).

## 3.6 Contributing your changes to *pandas*

### 3.6.1 Committing your code

Keep style fixes to a separate commit to make your pull request more readable.

Once you've made changes, you can see them by typing:

```
git status
```

If you have created a new file, it is not being tracked by git. Add it by typing:

```
git add path/to/file-to-be-added.py
```

Doing 'git status' again should give something like:

```
# On branch shiny-new-feature
#
#       modified:   /relative/path/to/file-you-added.py
#
```

Finally, commit your changes to your local repository with an explanatory message. *Pandas* uses a convention for commit message prefixes and layout. Here are some common prefixes along with general guidelines for when to use them:

- ENH: Enhancement, new functionality
- BUG: Bug fix

- DOC: Additions/updates to documentation
- TST: Additions/updates to tests
- BLD: Updates to the build process/scripts
- PERF: Performance improvement
- CLN: Code cleanup

The following defines how a commit message should be structured. Please reference the relevant GitHub issues in your commit message using GH1234 or #1234. Either style is fine, but the former is generally preferred:

- a subject line with < 80 chars.
- One blank line.
- Optionally, a commit message body.

Now you can commit your changes in your local repository:

```
git commit -m
```

### 3.6.2 Pushing your changes

When you want your changes to appear publicly on your GitHub page, push your forked feature branch's commits:

```
git push origin shiny-new-feature
```

Here `origin` is the default name given to your remote repository on GitHub. You can see the remote repositories:

```
git remote -v
```

If you added the upstream repository as described above you will see something like:

```
origin  git@github.com:yourname/pandas.git (fetch)
origin  git@github.com:yourname/pandas.git (push)
upstream      git://github.com/pandas-dev/pandas.git (fetch)
upstream      git://github.com/pandas-dev/pandas.git (push)
```

Now your code is on GitHub, but it is not yet a part of the *pandas* project. For that to happen, a pull request needs to be submitted on GitHub.

### 3.6.3 Review your code

When you're ready to ask for a code review, file a pull request. Before you do, once again make sure that you have followed all the guidelines outlined in this document regarding code style, tests, performance tests, and documentation. You should also double check your branch changes against the branch it was based on:

1. Navigate to your repository on GitHub – <https://github.com/your-user-name/pandas>
2. Click on `Branches`
3. Click on the `Compare` button for your feature branch
4. Select the `base` and `compare` branches, if necessary. This will be `master` and `shiny-new-feature`, respectively.



### 3.6.4 Finally, make the pull request

If everything looks good, you are ready to make a pull request. A pull request is how code from a local repository becomes available to the GitHub community and can be looked at and eventually merged into the master version. This pull request and its associated changes will eventually be committed to the master branch and available in the next release. To submit a pull request:

1. Navigate to your repository on GitHub
2. Click on the `Pull Request` button
3. You can then click on `Commits` and `Files Changed` to make sure everything looks okay one last time
4. Write a description of your changes in the `Preview Discussion` tab
5. Click `Send Pull Request`.

This request then goes to the repository maintainers, and they will review the code.

### 3.6.5 Updating your pull request

Based on the review you get on your pull request, you will probably need to make some changes to the code. In that case, you can make them in your branch, add a new commit to that branch, push it to GitHub, and the pull request will be automatically updated. Pushing them to GitHub again is done by:

```
git push origin shiny-new-feature
```

This will automatically update your pull request with the latest code and restart the *Continuous Integration* tests.

Another reason you might need to update your pull request is to solve conflicts with changes that have been merged into the master branch since you opened your pull request.

To do this, you need to “merge upstream master” in your branch:

```
git checkout shiny-new-feature
git fetch upstream
git merge upstream/master
```

If there are no conflicts (or they could be fixed automatically), a file with a default commit message will open, and you can simply save and quit this file.

If there are merge conflicts, you need to solve those conflicts. See for example at <https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/> for an explanation on how to do this. Once the conflicts are merged and the files where the conflicts were solved are added, you can run `git commit` to save those fixes.

If you have uncommitted changes at the moment you want to update the branch with master, you will need to `stash` them prior to updating (see the [stash docs](#)). This will effectively store your changes and they can be reapplied after updating.

After the feature branch has been update locally, you can now update your pull request by pushing to the branch on GitHub:

```
git push origin shiny-new-feature
```

### 3.6.6 Delete your merged branch (optional)

Once your feature branch is accepted into upstream, you’ll probably want to get rid of the branch. First, merge upstream master into your branch so git knows it is safe to delete your branch:

```
git fetch upstream
git checkout master
git merge upstream/master
```

Then you can do:

```
git branch -d shiny-new-feature
```

Make sure you use a lower-case `-d`, or else git won't warn you if your feature branch has not actually been merged.

The branch will still exist on GitHub, so to delete it there do:

```
git push origin --delete shiny-new-feature
```

## PACKAGE OVERVIEW

*pandas* is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the *Python* programming language.

*pandas* consists of the following elements:

- A set of labeled array data structures, the primary of which are Series and DataFrame.
- Index objects enabling both simple axis indexing and multi-level / hierarchical axis indexing.
- An integrated group by engine for aggregating and transforming data sets.
- Date range generation (`date_range`) and custom date offsets enabling the implementation of customized frequencies.
- Input/Output tools: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading pandas objects from the fast and efficient PyTables/HDF5 format.
- Memory-efficient “sparse” versions of the standard data structures for storing data that is mostly missing or mostly constant (some fixed value).
- Moving window statistics (rolling mean, rolling standard deviation, etc.).

### 4.1 Data Structures

| Dimensions | Name      | Description  |
|------------|-----------|--|
| 1          | Series    | 1D labeled homogeneously-typed array   |
| 2          | DataFrame | General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed column |

#### 4.1.1 Why more than one data structure?

The best way to think about the pandas data structures is as flexible containers for lower dimensional data. For example, DataFrame is a container for Series, and Series is a container for scalars. We would like to be able to insert and remove objects from these containers in a dictionary-like fashion.

Also, we would like sensible default behaviors for the common API functions which take into account the typical orientation of time series and cross-sectional data sets. When using ndarrays to store 2- and 3-dimensional data, a burden is placed on the user to consider the orientation of the data set when writing functions; axes are considered more or less equivalent (except when C- or Fortran-contiguity matters for performance). In pandas, the axes are intended to lend more semantic meaning to the data; i.e., for a particular data set there is likely to be a “right” way to orient the data. The goal, then, is to reduce the amount of mental effort required to code up data transformations in downstream functions.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1. Iterating through the columns of the DataFrame thus results in more readable code:

```
for col in df.columns:
    series = df[col]
    # do something with series
```

## 4.2 Mutability and copying of data

All pandas data structures are value-mutable (the values they contain can be altered) but not always size-mutable. The length of a Series cannot be changed, but, for example, columns can be inserted into a DataFrame. However, the vast majority of methods produce new objects and leave the input data untouched. In general we like to **favor immutability** where sensible.

## 4.3 Getting Support

The first stop for pandas issues and ideas is the [Github Issue Tracker](#). If you have a general question, pandas community experts can answer through [Stack Overflow](#).

## 4.4 Community

pandas is actively supported today by a community of like-minded individuals around the world who contribute their valuable time and energy to help make open source pandas possible. Thanks to [all of our contributors](#).

If you're interested in contributing, please visit [Contributing to pandas webpage](#).

pandas is a [NumFOCUS](#) sponsored project. This will help ensure the success of development of pandas as a world-class open-source project, and makes it possible to [donate](#) to the project.

## 4.5 Project Governance

The governance process that pandas project has used informally since its inception in 2008 is formalized in [Project Governance documents](#). The documents clarify how decisions are made and how the various elements of our community interact, including the relationship between open source collaborative development and work that may be funded by for-profit or non-profit entities.

Wes McKinney is the Benevolent Dictator for Life (BDFL).

## 4.6 Development Team

The list of the Core Team members and more detailed information can be found on the [people's page](#) of the governance repo.

## 4.7 Institutional Partners

The information about current institutional partners can be found on [pandas website page](#).

## 4.8 License

BSD 3-Clause License

Copyright (c) 2008–2012, AQR Capital Management, LLC, Lambda Foundry, Inc. and PyData\_↵  
Development Team  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- \* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- \* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- \* Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.



## 10 MINUTES TO PANDAS

This is a short introduction to pandas, geared mainly for new users. You can see more complex recipes in the *Cookbook*. Customarily, we import as follows:

```
In [1]: import pandas as pd
In [2]: import numpy as np
In [3]: import matplotlib.pyplot as plt
```

### 5.1 Object Creation

See the *Data Structure Intro section*.

Creating a *Series* by passing a list of values, letting pandas create a default integer index:

```
In [4]: s = pd.Series([1,3,5,np.nan,6,8])
In [5]: s
Out[5]:
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

Creating a *DataFrame* by passing a NumPy array, with a datetime index and labeled columns:

```
In [6]: dates = pd.date_range('20130101', periods=6)
In [7]: dates
Out[7]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
In [8]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
In [9]: df
Out[9]:
```

(continues on next page)

(continued from previous page)

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-01 | 0.469112  | -0.282863 | -1.509059 | -1.135632 |
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | -1.044236 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 1.071804  |
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 0.271860  |
| 2013-01-05 | -0.424972 | 0.567020  | 0.276232  | -1.087401 |
| 2013-01-06 | -0.673690 | 0.113648  | -1.478427 | 0.524988  |

Creating a DataFrame by passing a dict of objects that can be converted to series-like.

```
In [10]: df2 = pd.DataFrame({ 'A' : 1.,
.....:                      'B' : pd.Timestamp('20130102'),
.....:                      'C' : pd.Series(1, index=list(range(4)), dtype='float32'),
.....:                      'D' : np.array([3] * 4, dtype='int32'),
.....:                      'E' : pd.Categorical(["test", "train", "test", "train"]),
.....:                      'F' : 'foo' })
```

```
In [11]: df2
```

```
Out[11]:
```

|   | A   | B          | C   | D | E     | F   |
|---|-----|------------|-----|---|-------|-----|
| 0 | 1.0 | 2013-01-02 | 1.0 | 3 | test  | foo |
| 1 | 1.0 | 2013-01-02 | 1.0 | 3 | train | foo |
| 2 | 1.0 | 2013-01-02 | 1.0 | 3 | test  | foo |
| 3 | 1.0 | 2013-01-02 | 1.0 | 3 | train | foo |

The columns of the resulting DataFrame have different *dtypes*.

```
In [12]: df2.dtypes
```

```
Out[12]:
```

|        |                |
|--------|----------------|
| A      | float64        |
| B      | datetime64[ns] |
| C      | float32        |
| D      | int32          |
| E      | category       |
| F      | object         |
| dtype: | object         |

If you're using IPython, tab completion for column names (as well as public attributes) is automatically enabled. Here's a subset of the attributes that will be completed:

```
In [13]: df2.<TAB>
df2.A          df2.bool
df2.abs        df2.boxplot
df2.add        df2.C
df2.add_prefix df2.clip
df2.add_suffix df2.clip_lower
df2.align      df2.clip_upper
df2.all        df2.columns
df2.any        df2.combine
df2.append     df2.combine_first
df2.apply      df2.compound
df2.applymap   df2.consolidate
df2.D
```

As you can see, the columns A, B, C, and D are automatically tab completed. E is there as well; the rest of the attributes have been truncated for brevity.



## 5.2 Viewing Data

See the *Basics section*.

Here is how to view the top and bottom rows of the frame:

```
In [14]: df.head()
Out[14]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-01 | 0.469112  | -0.282863 | -1.509059 | -1.135632 |
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | -1.044236 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 1.071804  |
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 0.271860  |
| 2013-01-05 | -0.424972 | 0.567020  | 0.276232  | -1.087401 |

```
In [15]: df.tail(3)
Out[15]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 0.271860  |
| 2013-01-05 | -0.424972 | 0.567020  | 0.276232  | -1.087401 |
| 2013-01-06 | -0.673690 | 0.113648  | -1.478427 | 0.524988  |

Display the index, columns, and the underlying NumPy data:

```
In [16]: df.index
Out[16]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')

In [17]: df.columns
Out[17]:
Index(['A', 'B', 'C', 'D'], dtype='object')

In [18]: df.values
Out[18]:
array([[ 0.4691, -0.2829, -1.5091, -1.1356],
       [ 1.2121, -0.1732,  0.1192, -1.0442],
       [-0.8618, -2.1046, -0.4949,  1.0718],
       [ 0.7216, -0.7068, -1.0396,  0.2719],
       [-0.425 ,  0.567 ,  0.2762, -1.0874],
       [-0.6737,  0.1136, -1.4784,  0.525 ]])
```

`describe()` shows a quick statistic summary of your data:

```
In [19]: df.describe()
Out[19]:
```

|       | A         | B         | C         | D         |
|-------|-----------|-----------|-----------|-----------|
| count | 6.000000  | 6.000000  | 6.000000  | 6.000000  |
| mean  | 0.073711  | -0.431125 | -0.687758 | -0.233103 |
| std   | 0.843157  | 0.922818  | 0.779887  | 0.973118  |
| min   | -0.861849 | -2.104569 | -1.509059 | -1.135632 |
| 25%   | -0.611510 | -0.600794 | -1.368714 | -1.076610 |
| 50%   | 0.022070  | -0.228039 | -0.767252 | -0.386188 |
| 75%   | 0.658444  | 0.041933  | -0.034326 | 0.461706  |
| max   | 1.212112  | 0.567020  | 0.276232  | 1.071804  |

Transposing your data:

```
In [20]: df.T
Out[20]:
```

|   | 2013-01-01 | 2013-01-02 | 2013-01-03 | 2013-01-04 | 2013-01-05 | 2013-01-06 |
|---|------------|------------|------------|------------|------------|------------|
| A | 0.469112   | 1.212112   | -0.861849  | 0.721555   | -0.424972  | -0.673690  |
| B | -0.282863  | -0.173215  | -2.104569  | -0.706771  | 0.567020   | 0.113648   |
| C | -1.509059  | 0.119209   | -0.494929  | -1.039575  | 0.276232   | -1.478427  |
| D | -1.135632  | -1.044236  | 1.071804   | 0.271860   | -1.087401  | 0.524988   |

Sorting by an axis:

```
In [21]: df.sort_index(axis=1, ascending=False)
Out[21]:
```

|            | D         | C         | B         | A         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-01 | -1.135632 | -1.509059 | -0.282863 | 0.469112  |
| 2013-01-02 | -1.044236 | 0.119209  | -0.173215 | 1.212112  |
| 2013-01-03 | 1.071804  | -0.494929 | -2.104569 | -0.861849 |
| 2013-01-04 | 0.271860  | -1.039575 | -0.706771 | 0.721555  |
| 2013-01-05 | -1.087401 | 0.276232  | 0.567020  | -0.424972 |
| 2013-01-06 | 0.524988  | -1.478427 | 0.113648  | -0.673690 |

Sorting by values:

```
In [22]: df.sort_values(by='B')
Out[22]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 1.071804  |
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 0.271860  |
| 2013-01-01 | 0.469112  | -0.282863 | -1.509059 | -1.135632 |
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | -1.044236 |
| 2013-01-06 | -0.673690 | 0.113648  | -1.478427 | 0.524988  |
| 2013-01-05 | -0.424972 | 0.567020  | 0.276232  | -1.087401 |

## 5.3 Selection

---

**Note:** While standard Python / Numpy expressions for selecting and setting are intuitive and come in handy for interactive work, for production code, we recommend the optimized pandas data access methods, `.at`, `.iat`, `.loc` and `.iloc`.

---

See the indexing documentation [Indexing and Selecting Data](#) and [MultiIndex / Advanced Indexing](#).

### 5.3.1 Getting

Selecting a single column, which yields a `Series`, equivalent to `df.A`:

```
In [23]: df['A']
Out[23]:
```

|            |           |
|------------|-----------|
| 2013-01-01 | 0.469112  |
| 2013-01-02 | 1.212112  |
| 2013-01-03 | -0.861849 |
| 2013-01-04 | 0.721555  |

(continues on next page)

```
2013-01-05    -0.424972
2013-01-06    -0.673690
Freq: D, Name: A, dtype: float64
```

Out[24]:

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-01 | 0.469112  | -0.282863 | -1.509059 | -1.135632 |
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | -1.044236 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 1.071804  |

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | -1.044236 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 1.071804  |
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 0.271860  |

Out[26]:

|   |           |
|---|-----------|
| A | 0.469112  |
| B | -0.282863 |
| C | -1.509059 |
| D | -1.135632 |

Out [27]:

|            | A         | B         |
|------------|-----------|-----------|
| 2013-01-01 | 0.469112  | -0.282863 |
| 2013-01-02 | 1.212112  | -0.173215 |
| 2013-01-03 | -0.861849 | -2.104569 |
| 2013-01-04 | 0.721555  | -0.706771 |
| 2013-01-05 | -0.424972 | 0.567020  |
| 2013-01-06 | -0.673690 | 0.113648  |

Out [28]:

|            | A         | B         |
|------------|-----------|-----------|
| 2013-01-02 | 1.212112  | -0.173215 |
| 2013-01-03 | -0.861849 | -2.104569 |
| 2013-01-04 | 0.721555  | -0.706771 |

Reduction in the dimensions of the returned object:

```
In [29]: df.loc['20130102', ['A', 'B']]
Out[29]:
A      1.212112
B     -0.173215
Name: 2013-01-02 00:00:00, dtype: float64
```

For getting a scalar value:

```
In [30]: df.loc[dates[0], 'A']
Out[30]: 0.46911229990718628
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [31]: df.at[dates[0], 'A']
Out[31]: 0.46911229990718628
```

### 5.3.3 Selection by Position

See more in *Selection by Position*.

Select via the position of the passed integers:

```
In [32]: df.iloc[3]
Out[32]:
A      0.721555
B     -0.706771
C     -1.039575
D      0.271860
Name: 2013-01-04 00:00:00, dtype: float64
```

By integer slices, acting similar to numpy/python:

```
In [33]: df.iloc[3:5, 0:2]
Out[33]:
           A          B
2013-01-04  0.721555 -0.706771
2013-01-05 -0.424972  0.567020
```

By lists of integer position locations, similar to the numpy/python style:

```
In [34]: df.iloc[[1, 2, 4], [0, 2]]
Out[34]:
           A          C
2013-01-02  1.212112  0.119209
2013-01-03 -0.861849 -0.494929
2013-01-05 -0.424972  0.276232
```

For slicing rows explicitly:

```
In [35]: df.iloc[1:3, :]
Out[35]:
           A          B          C          D
2013-01-02  1.212112 -0.173215  0.119209 -1.044236
2013-01-03 -0.861849 -2.104569 -0.494929  1.071804
```

For slicing columns explicitly:

```
In [36]: df.iloc[:,1:3]
Out[36]:
```

|            | B         | C         |
|------------|-----------|-----------|
| 2013-01-01 | -0.282863 | -1.509059 |
| 2013-01-02 | -0.173215 | 0.119209  |
| 2013-01-03 | -2.104569 | -0.494929 |
| 2013-01-04 | -0.706771 | -1.039575 |
| 2013-01-05 | 0.567020  | 0.276232  |
| 2013-01-06 | 0.113648  | -1.478427 |

For getting a value explicitly:

```
In [37]: df.iloc[1,1]
Out[37]: -0.17321464905330858
```

For getting fast access to a scalar (equivalent to the prior method):

```
In [38]: df.iat[1,1]
Out[38]: -0.17321464905330858
```

### 5.3.4 Boolean Indexing

Using a single column's values to select data.

```
In [39]: df[df.A > 0]
Out[39]:
```

|            | A        | B         | C         | D         |
|------------|----------|-----------|-----------|-----------|
| 2013-01-01 | 0.469112 | -0.282863 | -1.509059 | -1.135632 |
| 2013-01-02 | 1.212112 | -0.173215 | 0.119209  | -1.044236 |
| 2013-01-04 | 0.721555 | -0.706771 | -1.039575 | 0.271860  |

Selecting values from a DataFrame where a boolean condition is met.

```
In [40]: df[df > 0]
Out[40]:
```

|            | A        | B        | C        | D        |
|------------|----------|----------|----------|----------|
| 2013-01-01 | 0.469112 | NaN      | NaN      | NaN      |
| 2013-01-02 | 1.212112 | NaN      | 0.119209 | NaN      |
| 2013-01-03 | NaN      | NaN      | NaN      | 1.071804 |
| 2013-01-04 | 0.721555 | NaN      | NaN      | 0.271860 |
| 2013-01-05 | NaN      | 0.567020 | 0.276232 | NaN      |
| 2013-01-06 | NaN      | 0.113648 | NaN      | 0.524988 |

Using the `isin()` method for filtering:

```
In [41]: df2 = df.copy()

In [42]: df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']

In [43]: df2
Out[43]:
```

|            | A        | B         | C         | D         | E   |
|------------|----------|-----------|-----------|-----------|-----|
| 2013-01-01 | 0.469112 | -0.282863 | -1.509059 | -1.135632 | one |
| 2013-01-02 | 1.212112 | -0.173215 | 0.119209  | -1.044236 | one |

(continues on next page)

(continued from previous page)

```

2013-01-03 -0.861849 -2.104569 -0.494929 1.071804 two
2013-01-04 0.721555 -0.706771 -1.039575 0.271860 three
2013-01-05 -0.424972 0.567020 0.276232 -1.087401 four
2013-01-06 -0.673690 0.113648 -1.478427 0.524988 three

```

```
In [44]: df2[df2['E'].isin(['two', 'four'])]
```

```

////////////////////////////////////
↪
      A      B      C      D      E
2013-01-03 -0.861849 -2.104569 -0.494929 1.071804 two
2013-01-05 -0.424972 0.567020 0.276232 -1.087401 four

```

### 5.3.5 Setting

Setting a new column automatically aligns the data by the indexes.

```
In [45]: s1 = pd.Series([1,2,3,4,5,6], index=pd.date_range('20130102', periods=6))
```

```
In [46]: s1
```

```
Out[46]:
```

```

2013-01-02    1
2013-01-03    2
2013-01-04    3
2013-01-05    4
2013-01-06    5
2013-01-07    6
Freq: D, dtype: int64

```

```
In [47]: df['F'] = s1
```

Setting values by label:

```
In [48]: df.at[dates[0], 'A'] = 0
```

Setting values by position:

```
In [49]: df.iat[0,1] = 0
```

Setting by assigning with a NumPy array:

```
In [50]: df.loc[:, 'D'] = np.array([5] * len(df))
```

The result of the prior setting operations.

```
In [51]: df
```

```
Out[51]:
```

```

      A      B      C      D      F
2013-01-01  0.000000  0.000000 -1.509059  5  NaN
2013-01-02  1.212112 -0.173215  0.119209  5  1.0
2013-01-03 -0.861849 -2.104569 -0.494929  5  2.0
2013-01-04  0.721555 -0.706771 -1.039575  5  3.0
2013-01-05 -0.424972  0.567020  0.276232  5  4.0
2013-01-06 -0.673690  0.113648 -1.478427  5  5.0

```

A where operation with setting.

```
In [52]: df2 = df.copy()

In [53]: df2[df2 > 0] = -df2

In [54]: df2
Out[54]:
```

|            | A         | B         | C         | D  | F    |
|------------|-----------|-----------|-----------|----|------|
| 2013-01-01 | 0.000000  | 0.000000  | -1.509059 | -5 | NaN  |
| 2013-01-02 | -1.212112 | -0.173215 | -0.119209 | -5 | -1.0 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | -5 | -2.0 |
| 2013-01-04 | -0.721555 | -0.706771 | -1.039575 | -5 | -3.0 |
| 2013-01-05 | -0.424972 | -0.567020 | -0.276232 | -5 | -4.0 |
| 2013-01-06 | -0.673690 | -0.113648 | -1.478427 | -5 | -5.0 |

## 5.4 Missing Data

pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the [Missing Data section](#).

Reindexing allows you to change/add/delete the index on a specified axis. This returns a copy of the data.

```
In [55]: df1 = df.reindex(index=dates[0:4], columns=list(df.columns) + ['E'])

In [56]: df1.loc[dates[0]:dates[1], 'E'] = 1

In [57]: df1
Out[57]:
```

|            | A         | B         | C         | D | F   | E   |
|------------|-----------|-----------|-----------|---|-----|-----|
| 2013-01-01 | 0.000000  | 0.000000  | -1.509059 | 5 | NaN | 1.0 |
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | 5 | 1.0 | 1.0 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 5 | 2.0 | NaN |
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 5 | 3.0 | NaN |

To drop any rows that have missing data.

```
In [58]: df1.dropna(how='any')
Out[58]:
```

|            | A        | B         | C        | D | F   | E   |
|------------|----------|-----------|----------|---|-----|-----|
| 2013-01-02 | 1.212112 | -0.173215 | 0.119209 | 5 | 1.0 | 1.0 |

Filling missing data.

```
In [59]: df1.fillna(value=5)
Out[59]:
```

|            | A         | B         | C         | D | F   | E   |
|------------|-----------|-----------|-----------|---|-----|-----|
| 2013-01-01 | 0.000000  | 0.000000  | -1.509059 | 5 | 5.0 | 1.0 |
| 2013-01-02 | 1.212112  | -0.173215 | 0.119209  | 5 | 1.0 | 1.0 |
| 2013-01-03 | -0.861849 | -2.104569 | -0.494929 | 5 | 2.0 | 5.0 |
| 2013-01-04 | 0.721555  | -0.706771 | -1.039575 | 5 | 3.0 | 5.0 |

To get the boolean mask where values are nan.

```
In [60]: pd.isna(df1)
Out[60]:
```

|  | A | B | C | D | F | E |
|--|---|---|---|---|---|---|
|--|---|---|---|---|---|---|

(continues on next page)

(continued from previous page)

|            |       |       |       |       |       |       |
|------------|-------|-------|-------|-------|-------|-------|
| 2013-01-01 | False | False | False | False | True  | False |
| 2013-01-02 | False | False | False | False | False | False |
| 2013-01-03 | False | False | False | False | False | True  |
| 2013-01-04 | False | False | False | False | False | True  |

## 5.5 Operations

See the *Basic section on Binary Ops*.

### 5.5.1 Stats

Operations in general *exclude* missing data.

Performing a descriptive statistic:

```
In [61]: df.mean()
Out[61]:
A    -0.004474
B    -0.383981
C    -0.687758
D     5.000000
F     3.000000
dtype: float64
```

Same operation on the other axis:

```
In [62]: df.mean(1)
Out[62]:
2013-01-01    0.872735
2013-01-02    1.431621
2013-01-03    0.707731
2013-01-04    1.395042
2013-01-05    1.883656
2013-01-06    1.592306
Freq: D, dtype: float64
```

Operating with objects that have different dimensionality and need alignment. In addition, pandas automatically broadcasts along the specified dimension.

```
In [63]: s = pd.Series([1,3,5,np.nan,6,8], index=dates).shift(2)
In [64]: s
Out[64]:
2013-01-01    NaN
2013-01-02    NaN
2013-01-03     1.0
2013-01-04     3.0
2013-01-05     5.0
2013-01-06    NaN
Freq: D, dtype: float64
```

```
In [65]: df.sub(s, axis='index')
```

(continues on next page)





(continued from previous page)

```

////////////////////////////////////Out [70]
↪
4      5
6      2
2      2
1      1
dtype: int64

```

## 5.5.4 String Methods

Series is equipped with a set of string processing methods in the *str* attribute that make it easy to operate on each element of the array, as in the code snippet below. Note that pattern-matching in *str* generally uses [regular expressions](#) by default (and in some cases always uses them). See more at [Vectorized String Methods](#).

```

In [71]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [72]: s.str.lower()
Out [72]:
0      a
1      b
2      c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object

```

## 5.6 Merge

### 5.6.1 Concat

pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

See the [Merging section](#).

Concatenating pandas objects together with `concat()`:

```

In [73]: df = pd.DataFrame(np.random.randn(10, 4))

In [74]: df
Out [74]:
   0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867

```

(continues on next page)

(continued from previous page)

```

7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495

# break it into pieces
In [75]: pieces = [df[:3], df[3:7], df[7:]]

In [76]: pd.concat(pieces)
Out[76]:
   0         1         2         3
0 -0.548702  1.467327 -1.015962 -0.483075
1  1.637550 -1.217659 -0.291519 -1.745505
2 -0.263952  0.991460 -0.919069  0.266046
3 -0.709661  1.669052  1.037882 -1.705775
4 -0.919854 -0.042379  1.247642 -0.009920
5  0.290213  0.495767  0.362949  1.548106
6 -1.131345 -0.089329  0.337863 -0.945867
7 -0.932132  1.956030  0.017587 -0.016692
8 -0.575247  0.254161 -1.143704  0.215897
9  1.193555 -0.077118 -0.408530 -0.862495

```

## 5.6.2 Join

SQL style merges. See the [Database style joining](#) section.

```

In [77]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})

In [78]: right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})

In [79]: left
Out[79]:
   key  lval
0  foo     1
1  foo     2

In [80]: right
Out[80]:
   key  rval
0  foo     4
1  foo     5

In [81]: pd.merge(left, right, on='key')
Out[81]:
   key  lval  rval
0  foo     1     4
1  foo     1     5
2  foo     2     4
3  foo     2     5

```

Another example that can be given is:

```

In [82]: left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})

In [83]: right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})

```

(continues on next page)

(continued from previous page)

```

In [84]: left
Out[84]:
   key  lval
0  foo     1
1  bar     2

In [85]: right
Out[85]:
   key  rval
0  foo     4
1  bar     5

In [86]: pd.merge(left, right, on='key')
Out[86]:
   key  lval  rval
0  foo     1     4
1  bar     2     5

```

### 5.6.3 Append

Append rows to a dataframe. See the *Appending* section.

```

In [87]: df = pd.DataFrame(np.random.randn(8, 4), columns=['A', 'B', 'C', 'D'])

In [88]: df
Out[88]:
   A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346  0.339969 -0.693205
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351  1.715071 -0.708758

In [89]: s = df.iloc[3]

In [90]: df.append(s, ignore_index=True)
Out[90]:
   A         B         C         D
0  1.346061  1.511763  1.627081 -0.990582
1 -0.441652  1.211526  0.268520  0.024580
2 -1.577585  0.396823 -0.105381 -0.532532
3  1.453749  1.208843 -0.080952 -0.264610
4 -0.727965 -0.589346  0.339969 -0.693205
5 -0.339355  0.593616  0.884345  1.591431
6  0.141809  0.220390  0.435589  0.192451
7 -0.096701  0.803351  1.715071 -0.708758
8  1.453749  1.208843 -0.080952 -0.264610

```

## 5.7 Grouping

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria
- **Applying** a function to each group independently
- **Combining** the results into a data structure

See the *Grouping section*.

```
In [91]: df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
....:                             'foo', 'bar', 'foo', 'foo'],
....:                      'B' : ['one', 'one', 'two', 'three',
....:                             'two', 'two', 'one', 'three'],
....:                      'C' : np.random.randn(8),
....:                      'D' : np.random.randn(8) })
....:
```

```
In [92]: df
Out[92]:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 0 | foo | one   | -1.202872 | -0.055224 |
| 1 | bar | one   | -1.814470 | 2.395985  |
| 2 | foo | two   | 1.018601  | 1.552825  |
| 3 | bar | three | -0.595447 | 0.166599  |
| 4 | foo | two   | 1.395433  | 0.047609  |
| 5 | bar | two   | -0.392670 | -0.136473 |
| 6 | foo | one   | 0.007207  | -0.561757 |
| 7 | foo | three | 1.928123  | -1.623033 |

Grouping and then applying the `sum()` function to the resulting groups.

```
In [93]: df.groupby('A').sum()
Out[93]:
```

|     | C         | D        |
|-----|-----------|----------|
| A   |           |          |
| bar | -2.802588 | 2.42611  |
| foo | 3.146492  | -0.63958 |

Grouping by multiple columns forms a hierarchical index, and again we can apply the `sum` function.

```
In [94]: df.groupby(['A', 'B']).sum()
Out[94]:
```

|     |       | C         | D         |
|-----|-------|-----------|-----------|
| A   | B     |           |           |
| bar | one   | -1.814470 | 2.395985  |
|     | three | -0.595447 | 0.166599  |
|     | two   | -0.392670 | -0.136473 |
| foo | one   | -1.195665 | -0.616981 |
|     | three | 1.928123  | -1.623033 |
|     | two   | 2.414034  | 1.600434  |

## 5.8 Reshaping

See the sections on *Hierarchical Indexing* and *Reshaping*.

### 5.8.1 Stack

```
In [95]: tuples = list(zip(*[['bar', 'bar', 'baz', 'baz'],
.....:                        ['foo', 'foo', 'qux', 'qux'],
.....:                        ['one', 'two', 'one', 'two'],
.....:                        ['one', 'two', 'one', 'two']]))
.....:

In [96]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [97]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])

In [98]: df2 = df[:4]

In [99]: df2
Out[99]:
```

|       |        | A         | B         |
|-------|--------|-----------|-----------|
| first | second |           |           |
| bar   | one    | 0.029399  | -0.542108 |
|       | two    | 0.282696  | -0.087302 |
| baz   | one    | -1.575170 | 1.771208  |
|       | two    | 0.816482  | 1.100230  |

The `stack()` method “compresses” a level in the DataFrame’s columns.

```
In [100]: stacked = df2.stack()

In [101]: stacked
Out[101]:
first second
bar      one      A      0.029399
          one      B     -0.542108
          two      A      0.282696
          two      B     -0.087302
baz      one      A     -1.575170
          one      B      1.771208
          two      A      0.816482
          two      B      1.100230
dtype: float64
```

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of `stack()` is `unstack()`, which by default unstacks the **last level**:

```
In [102]: stacked.unstack()
Out[102]:
```

|       |        | A         | B         |
|-------|--------|-----------|-----------|
| first | second |           |           |
| bar   | one    | 0.029399  | -0.542108 |
|       | two    | 0.282696  | -0.087302 |
| baz   | one    | -1.575170 | 1.771208  |
|       | two    | 0.816482  | 1.100230  |

```
In [103]: stacked.unstack(1)
////////////////////////////////////
```

|        | one | two |
|--------|-----|-----|
| second |     |     |
| first  |     |     |

(continues on next page)

(continued from previous page)

```
bar   A   0.029399   0.282696
      B  -0.542108  -0.087302
baz   A  -1.575170   0.816482
      B   1.771208   1.100230
```

```
In [104]: stacked.unstack(0)
```

```
////////////////////////////////////
```

```
↪
first      bar      baz
second
one   A   0.029399 -1.575170
      B  -0.542108   1.771208
two   A   0.282696   0.816482
      B  -0.087302   1.100230
```

## 5.8.2 Pivot Tables

See the section on *Pivot Tables*.

```
In [105]: df = pd.DataFrame({'A' : ['one', 'one', 'two', 'three'] * 3,
.....:                      'B' : ['A', 'B', 'C'] * 4,
.....:                      'C' : ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2,
.....:                      'D' : np.random.randn(12),
.....:                      'E' : np.random.randn(12)})
```

```
In [106]: df
```

```
Out[106]:
```

|    | A     | B | C   | D         | E         |
|----|-------|---|-----|-----------|-----------|
| 0  | one   | A | foo | 1.418757  | -0.179666 |
| 1  | one   | B | foo | -1.879024 | 1.291836  |
| 2  | two   | C | foo | 0.536826  | -0.009614 |
| 3  | three | A | bar | 1.006160  | 0.392149  |
| 4  | one   | B | bar | -0.029716 | 0.264599  |
| 5  | one   | C | bar | -1.146178 | -0.057409 |
| 6  | two   | A | foo | 0.100900  | -1.425638 |
| 7  | three | B | foo | -1.035018 | 1.024098  |
| 8  | one   | C | foo | 0.314665  | -0.106062 |
| 9  | one   | A | bar | -0.773723 | 1.824375  |
| 10 | two   | B | bar | -1.170653 | 0.595974  |
| 11 | three | C | bar | 0.648740  | 1.167115  |

We can produce pivot tables from this data very easily:

```
In [107]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
```

```
Out[107]:
```

|       |   | bar       | foo       |
|-------|---|-----------|-----------|
| A     |   |           |           |
| B     |   |           |           |
| one   | A | -0.773723 | 1.418757  |
|       | B | -0.029716 | -1.879024 |
|       | C | -1.146178 | 0.314665  |
| three |   |           |           |
|       | A | 1.006160  | NaN       |
|       | B | NaN       | -1.035018 |
|       | C | 0.648740  | NaN       |
| two   |   |           |           |
|       | A | NaN       | 0.100900  |

(continues on next page)

(continued from previous page)

|   |           |          |
|---|-----------|----------|
| B | -1.170653 | NaN      |
| C | NaN       | 0.536826 |

## 5.9 Time Series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications. See the *Time Series section*.

```
In [108]: rng = pd.date_range('1/1/2012', periods=100, freq='S')

In [109]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)

In [110]: ts.resample('5Min').sum()
Out[110]:
2012-01-01    25083
Freq: 5T, dtype: int64
```

Time zone representation:

```
In [111]: rng = pd.date_range('3/6/2012 00:00', periods=5, freq='D')

In [112]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [113]: ts
Out[113]:
2012-03-06    0.464000
2012-03-07    0.227371
2012-03-08   -0.496922
2012-03-09    0.306389
2012-03-10   -2.290613
Freq: D, dtype: float64

In [114]: ts_utc = ts.tz_localize('UTC')

In [115]: ts_utc
Out[115]:
2012-03-06 00:00:00+00:00    0.464000
2012-03-07 00:00:00+00:00    0.227371
2012-03-08 00:00:00+00:00   -0.496922
2012-03-09 00:00:00+00:00    0.306389
2012-03-10 00:00:00+00:00   -2.290613
Freq: D, dtype: float64
```

Converting to another time zone:

```
In [116]: ts_utc.tz_convert('US/Eastern')
Out[116]:
2012-03-05 19:00:00-05:00    0.464000
2012-03-06 19:00:00-05:00    0.227371
2012-03-07 19:00:00-05:00   -0.496922
2012-03-08 19:00:00-05:00    0.306389
2012-03-09 19:00:00-05:00   -2.290613
Freq: D, dtype: float64
```



```
In [118]: ts = pd.Series(np.random.randn(len(rng)), index=rng)
```

Out [119]:

```
In [120]: ps = ts.to_period()
```

Out [121] :

```
In [122]: ps.to_timestamp()
```



```
2012-01-01    -1.134623
2012-02-01    -1.561819
2012-03-01    -0.260838
2012-04-01     0.281957
2012-05-01     1.523962
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [124]: ts = pd.Series(np.random.randn(len(prng)), prng)
```

```
In [126]: ts.head()
```

Out [126] :

```
1990-03-01 09:00 -0.902937
1990-06-01 09:00 0.068159
1990-09-01 09:00 -0.057873
1990-12-01 09:00 -0.368204
1991-03-01 09:00 -1.144073
Freq: H, dtype: float64
```

## 5.10 Categoricals

pandas can include categorical data in a DataFrame. For full docs, see the [categorical introduction](#) and the [API documentation](#).

```
In [127]: df = pd.DataFrame({"id": [1, 2, 3, 4, 5, 6], "raw_grade": ['a', 'b', 'b', 'a', 'a',
↪ 'e']})
```

Convert the raw grades to a categorical data type.

```
In [128]: df["grade"] = df["raw_grade"].astype("category")
```

```
In [129]: df["grade"]
```

```
Out[129]:
```

```
0      a
1      b
2      b
3      a
4      a
5      e
Name: grade, dtype: category
Categories (3, object): [a, b, e]
```

Rename the categories to more meaningful names (assigning to `Series.cat.categories` is inplace!).

```
In [130]: df["grade"].cat.categories = ["very good", "good", "very bad"]
```

Reorder the categories and simultaneously add the missing categories (methods under `Series.cat` return a new Series by default).

```
In [131]: df["grade"] = df["grade"].cat.set_categories(["very bad", "bad", "medium",
↪ "good", "very good"])
```

```
In [132]: df["grade"]
```

```
Out[132]:
```

```
0      very good
1           good
2           good
3      very good
4      very good
5      very bad
Name: grade, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]
```

Sorting is per order in the categories, not lexical order.

```
In [133]: df.sort_values(by="grade")
```

```
Out[133]:
```

```
   id raw_grade  grade
5   6         e  very bad
1   2         b    good
2   3         b    good
0   1         a  very good
3   4         a  very good
4   5         a  very good
```

Grouping by a categorical column also shows empty categories.

```
In [134]: df.groupby("grade").size()
Out[134]:
grade
very bad    1
bad         0
medium      0
good        2
very good   3
dtype: int64
```

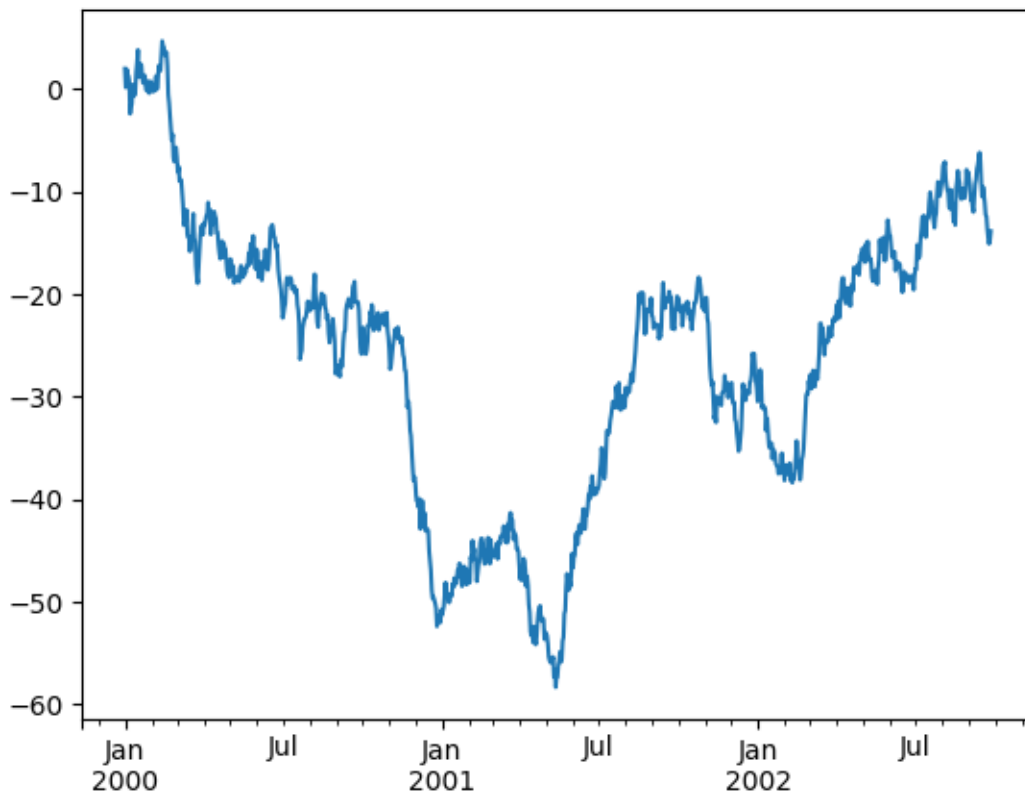
## 5.11 Plotting

See the [Plotting](#) docs.

```
In [135]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
↳ periods=1000))

In [136]: ts = ts.cumsum()

In [137]: ts.plot()
Out[137]: <matplotlib.axes._subplots.AxesSubplot at 0x1192135f8>
```



On a DataFrame, the `plot()` method is a convenience to plot all of the columns with labels:

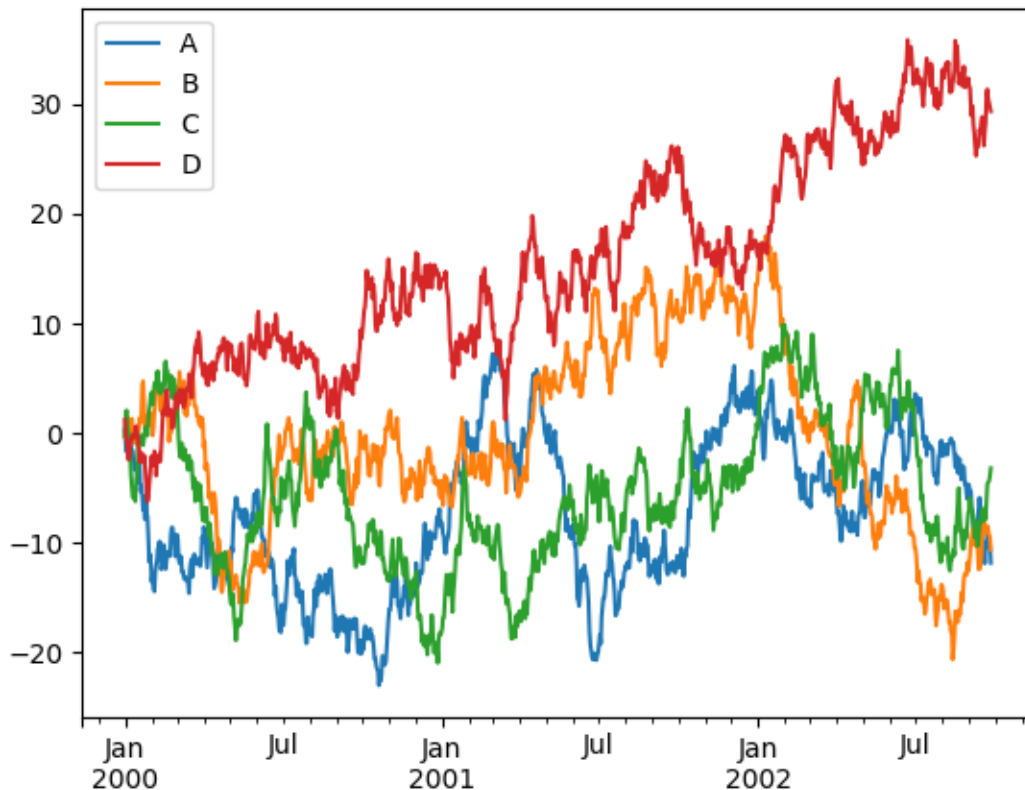
```

In [138]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,
.....:                      columns=['A', 'B', 'C', 'D'])
.....:

In [139]: df = df.cumsum()

In [140]: plt.figure(); df.plot(); plt.legend(loc='best')
Out[140]: <matplotlib.legend.Legend at 0x1234e1dd8>

```



## 5.12 Getting Data In/Out

### 5.12.1 CSV

*Writing to a csv file.*

```
In [141]: df.to_csv('foo.csv')
```

*Reading from a csv file.*

```
In [142]: pd.read_csv('foo.csv')
```

```

Out[142]:
   Unnamed: 0      A      B      C      D

```

(continues on next page)

(continued from previous page)

```

0      2000-01-01      0.266457 -0.399641 -0.219582      1.186860
1      2000-01-02     -1.170732 -0.345873      1.653061     -0.282953
2      2000-01-03     -1.734933      0.530468      2.060811     -0.515536
3      2000-01-04     -1.555121      1.452620      0.239859     -1.156896
4      2000-01-05      0.578117      0.511371      0.103552     -2.428202
5      2000-01-06      0.478344      0.449933     -0.741620     -1.962409
6      2000-01-07      1.235339     -0.091757     -1.543861     -1.084753
...      ...      ...      ...      ...      ...
993    2002-09-20    -10.628548     -9.153563     -7.883146     28.313940
994    2002-09-21    -10.390377     -8.727491     -6.399645     30.914107
995    2002-09-22     -8.985362     -8.485624     -4.669462     31.367740
996    2002-09-23     -9.558560     -8.781216     -4.499815     30.518439
997    2002-09-24     -9.902058     -9.340490     -4.386639     30.105593
998    2002-09-25    -10.216020     -9.480682     -3.933802     29.758560
999    2002-09-26    -11.856774    -10.671012     -3.216025     29.369368

[1000 rows x 5 columns]
```

### 5.12.2 HDF5

Reading and writing to *HDFStores*.

Writing to a HDF5 Store.

```
In [143]: df.to_hdf('foo.h5', 'df')
```

Reading from a HDF5 Store.

```

In [144]: pd.read_hdf('foo.h5', 'df')
Out[144]:
```

|            | A          | B          | C         | D         |
|------------|------------|------------|-----------|-----------|
| 2000-01-01 | 0.266457   | -0.399641  | -0.219582 | 1.186860  |
| 2000-01-02 | -1.170732  | -0.345873  | 1.653061  | -0.282953 |
| 2000-01-03 | -1.734933  | 0.530468   | 2.060811  | -0.515536 |
| 2000-01-04 | -1.555121  | 1.452620   | 0.239859  | -1.156896 |
| 2000-01-05 | 0.578117   | 0.511371   | 0.103552  | -2.428202 |
| 2000-01-06 | 0.478344   | 0.449933   | -0.741620 | -1.962409 |
| 2000-01-07 | 1.235339   | -0.091757  | -1.543861 | -1.084753 |
| ...        | ...        | ...        | ...       | ...       |
| 2002-09-20 | -10.628548 | -9.153563  | -7.883146 | 28.313940 |
| 2002-09-21 | -10.390377 | -8.727491  | -6.399645 | 30.914107 |
| 2002-09-22 | -8.985362  | -8.485624  | -4.669462 | 31.367740 |
| 2002-09-23 | -9.558560  | -8.781216  | -4.499815 | 30.518439 |
| 2002-09-24 | -9.902058  | -9.340490  | -4.386639 | 30.105593 |
| 2002-09-25 | -10.216020 | -9.480682  | -3.933802 | 29.758560 |
| 2002-09-26 | -11.856774 | -10.671012 | -3.216025 | 29.369368 |

```

[1000 rows x 4 columns]
```

### 5.12.3 Excel

Reading and writing to *MS Excel*.

Writing to an excel file.

```
In [145]: df.to_excel('foo.xlsx', sheet_name='Sheet1')
```

Reading from an excel file.

```
In [146]: pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA'])
```

Out[146]:

|            | A          | B          | C         | D         |
|------------|------------|------------|-----------|-----------|
| 2000-01-01 | 0.266457   | -0.399641  | -0.219582 | 1.186860  |
| 2000-01-02 | -1.170732  | -0.345873  | 1.653061  | -0.282953 |
| 2000-01-03 | -1.734933  | 0.530468   | 2.060811  | -0.515536 |
| 2000-01-04 | -1.555121  | 1.452620   | 0.239859  | -1.156896 |
| 2000-01-05 | 0.578117   | 0.511371   | 0.103552  | -2.428202 |
| 2000-01-06 | 0.478344   | 0.449933   | -0.741620 | -1.962409 |
| 2000-01-07 | 1.235339   | -0.091757  | -1.543861 | -1.084753 |
| ...        | ...        | ...        | ...       | ...       |
| 2002-09-20 | -10.628548 | -9.153563  | -7.883146 | 28.313940 |
| 2002-09-21 | -10.390377 | -8.727491  | -6.399645 | 30.914107 |
| 2002-09-22 | -8.985362  | -8.485624  | -4.669462 | 31.367740 |
| 2002-09-23 | -9.558560  | -8.781216  | -4.499815 | 30.518439 |
| 2002-09-24 | -9.902058  | -9.340490  | -4.386639 | 30.105593 |
| 2002-09-25 | -10.216020 | -9.480682  | -3.933802 | 29.758560 |
| 2002-09-26 | -11.856774 | -10.671012 | -3.216025 | 29.369368 |

[1000 rows x 4 columns]

## 5.13 Gotchas

If you are attempting to perform an operation you might see an exception like:

```
>>> if pd.Series([False, True, False]):
    print("I was true")
Traceback
...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

See [Comparisons](#) for an explanation and what to do.

See [Gotchas](#) as well.

## TUTORIALS

This is a guide to many pandas tutorials, geared mainly for new users.

### 6.1 Internal Guides

pandas' own *10 Minutes to pandas*.

More complex recipes are in the *Cookbook*.

A handy pandas [cheat sheet](#).

### 6.2 pandas Cookbook

The goal of this 2015 cookbook (by [Julia Evans](#)) is to give you some concrete examples for getting started with pandas. These are examples with real-world data, and all the bugs and weirdness that entails.

Here are links to the v0.2 release. For an up-to-date table of contents, see the [pandas-cookbook GitHub repository](#). To run the examples in this tutorial, you'll need to clone the GitHub repository and get IPython Notebook running. See [How to use this cookbook](#).

- [A quick tour of the IPython Notebook](#): Shows off IPython's awesome tab completion and magic functions.
- [Chapter 1](#): Reading your data into pandas is pretty much the easiest thing. Even when the encoding is wrong!
- [Chapter 2](#): It's not totally obvious how to select data from a pandas dataframe. Here we explain the basics (how to take slices and get columns)
- [Chapter 3](#): Here we get into serious slicing and dicing and learn how to filter dataframes in complicated ways, really fast.
- [Chapter 4](#): Groupby/aggregate is seriously my favorite thing about pandas and I use it all the time. You should probably read this.
- [Chapter 5](#): Here you get to find out if it's cold in Montreal in the winter (spoiler: yes). Web scraping with pandas is fun! Here we combine dataframes.
- [Chapter 6](#): Strings with pandas are great. It has all these vectorized string operations and they're the best. We will turn a bunch of strings containing "Snow" into vectors of numbers in a trice.
- [Chapter 7](#): Cleaning up messy data is never a joy, but with pandas it's easier.
- [Chapter 8](#): Parsing Unix timestamps is confusing at first but it turns out to be really easy.
- [Chapter 9](#): Reading data from SQL databases.

## 6.3 Lessons for new pandas users

For more resources, please visit the main [repository](#).

- **01 - Lesson:** - Importing libraries - Creating data sets - Creating data frames - Reading from CSV - Exporting to CSV - Finding maximums - Plotting data
- **02 - Lesson:** - Reading from TXT - Exporting to TXT - Selecting top/bottom records - Descriptive statistics - Grouping/sorting data
- **03 - Lesson:** - Creating functions - Reading from EXCEL - Exporting to EXCEL - Outliers - Lambda functions - Slice and dice data
- **04 - Lesson:** - Adding/deleting columns - Index operations
- **05 - Lesson:** - Stack/Unstack/Transpose functions
- **06 - Lesson:** - GroupBy function
- **07 - Lesson:** - Ways to calculate outliers
- **08 - Lesson:** - Read from Microsoft SQL databases
- **09 - Lesson:** - Export to CSV/EXCEL/TXT
- **10 - Lesson:** - Converting between different kinds of formats
- **11 - Lesson:** - Combining data from various sources

## 6.4 Practical data analysis with Python

This [guide](#) is a comprehensive introduction to the data analysis process using the Python data ecosystem and an interesting open dataset. There are four sections covering selected topics as follows:

- [Munging Data](#)
- [Aggregating Data](#)
- [Visualizing Data](#)
- [Time Series](#)

## 6.5 Exercises for new users

Practice your skills with real data sets and exercises. For more resources, please visit the main [repository](#).

- [01 - Getting & Knowing Your Data](#)
- [02 - Filtering & Sorting](#)
- [03 - Grouping](#)
- [04 - Apply](#)
- [05 - Merge](#)
- [06 - Stats](#)
- [07 - Visualization](#)
- [08 - Creating Series and DataFrames](#)



- [09 - Time Series](#)
- [10 - Deleting](#)

## 6.6 Modern pandas

Tutorial series written in 2016 by [Tom Augspurger](#). The source may be found in the GitHub repository [TomAugspurger/effective-pandas](#).

- [Modern Pandas](#)
- [Method Chaining](#)
- [Indexes](#)
- [Performance](#)
- [Tidy Data](#)
- [Visualization](#)
- [Timeseries](#)

## 6.7 Excel charts with pandas, vincent and xlsxwriter

- [Using Pandas and XlsxWriter to create Excel charts](#)

## 6.8 Video Tutorials

- [Pandas From The Ground Up \(2015\) \(2:24\) GitHub repo](#)
- [Introduction Into Pandas \(2016\) \(1:28\) GitHub repo](#)
- [Pandas: .head\(\) to .tail\(\) \(2016\) \(1:26\) GitHub repo](#)

## 6.9 Various Tutorials

- [Wes McKinney's \(pandas BDFL\) blog](#)
- [Statistical analysis made easy in Python with SciPy and pandas DataFrames](#), by Randal Olson
- [Statistical Data Analysis in Python](#), tutorial videos, by Christopher Fonnesbeck from SciPy 2013
- [Financial analysis in Python](#), by Thomas Wiecki
- [Intro to pandas data structures](#), by Greg Reda
- [Pandas and Python: Top 10](#), by Manish Amde
- [Pandas Tutorial](#), by Mikhail Semeniuk
- [Pandas DataFrames Tutorial](#), by Karlijn Willems
- [A concise tutorial with real life examples](#)



## COOKBOOK

This is a repository for *short and sweet* examples and links for useful pandas recipes. We encourage users to add to this documentation.

Adding interesting links and/or inline examples to this section is a great *First Pull Request*.

Simplified, condensed, new-user friendly, in-line examples have been inserted where possible to augment the Stack-Overflow and GitHub links. Many of the links contain expanded information, above what the in-line examples offer.

Pandas (pd) and Numpy (np) are the only two abbreviated imported modules. The rest are kept explicitly imported for newer users.

These examples are written for Python 3. Minor tweaks might be necessary for earlier python versions.

### 7.1 Idioms

These are some neat pandas idioms

if-then/if-then-else on one column, and assignment to another one or more columns:

```
In [1]: df = pd.DataFrame(
...:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]})
...:
Out[1]:
```

|   | AAA | BBB | CCC |
|---|-----|-----|-----|
| 0 | 4   | 10  | 100 |
| 1 | 5   | 20  | 50  |
| 2 | 6   | 30  | -30 |
| 3 | 7   | 40  | -50 |

#### 7.1.1 if-then...

An if-then on one column

```
In [2]: df.loc[df.AAA >= 5, 'BBB'] = -1; df
Out[2]:
```

|   | AAA | BBB | CCC |
|---|-----|-----|-----|
| 0 | 4   | 10  | 100 |
| 1 | 5   | -1  | 50  |
| 2 | 6   | -1  | -30 |
| 3 | 7   | -1  | -50 |

An if-then with assignment to 2 columns:

```
In [3]: df.loc[df.AAA >= 5, ['BBB', 'CCC']] = 555; df
Out[3]:
```

|   | AAA | BBB | CCC |
|---|-----|-----|-----|
| 0 | 4   | 10  | 100 |
| 1 | 5   | 555 | 555 |
| 2 | 6   | 555 | 555 |
| 3 | 7   | 555 | 555 |

Add another line with different logic, to do the -else

```
In [4]: df.loc[df.AAA < 5, ['BBB', 'CCC']] = 2000; df
Out[4]:
```

|   | AAA | BBB  | CCC  |
|---|-----|------|------|
| 0 | 4   | 2000 | 2000 |
| 1 | 5   | 555  | 555  |
| 2 | 6   | 555  | 555  |
| 3 | 7   | 555  | 555  |

Or use pandas where after you've set up a mask

```
In [5]: df_mask = pd.DataFrame({'AAA' : [True] * 4, 'BBB' : [False] * 4, 'CCC' : [True,
↪False] * 2})

In [6]: df.where(df_mask, -1000)
Out[6]:
```

|   | AAA | BBB   | CCC   |
|---|-----|-------|-------|
| 0 | 4   | -1000 | 2000  |
| 1 | 5   | -1000 | -1000 |
| 2 | 6   | -1000 | 555   |
| 3 | 7   | -1000 | -1000 |

if-then-else using numpy's where()

```
In [7]: df = pd.DataFrame(
...:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]})
...:
Out[7]:
```

|   | AAA | BBB | CCC |
|---|-----|-----|-----|
| 0 | 4   | 10  | 100 |
| 1 | 5   | 20  | 50  |
| 2 | 6   | 30  | -30 |
| 3 | 7   | 40  | -50 |

```
In [8]: df['logic'] = np.where(df['AAA'] > 5, 'high', 'low'); df
Out[8]:
```

|   | AAA | BBB | CCC | logic |
|---|-----|-----|-----|-------|
| 0 | 4   | 10  | 100 | low   |
| 1 | 5   | 20  | 50  | low   |
| 2 | 6   | 30  | -30 | high  |
| 3 | 7   | 40  | -50 | high  |

## 7.1.2 Splitting

Split a frame with a boolean criterion

```
In [9]: df = pd.DataFrame(
...:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}; df
...:
```

```
Out[9]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
2     6   30  -30
3     7   40  -50
```

```
In [10]: dflow = df[df.AAA <= 5]; dflow
```

```
Out[10]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
```

```
In [11]: dfhigh = df[df.AAA > 5]; dfhigh
```

```
Out[11]:
   AAA  BBB  CCC
2     6   30  -30
3     7   40  -50
```

### 7.1.3 Building Criteria

Select with multi-column criteria

```
In [12]: df = pd.DataFrame(
...:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}; df
...:
```

```
Out[12]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
2     6   30  -30
3     7   40  -50
```

... and (without assignment returns a Series)

```
In [13]: newseries = df.loc[(df['BBB'] < 25) & (df['CCC'] >= -40), 'AAA']; newseries
Out[13]:
0     4
1     5
Name: AAA, dtype: int64
```

... or (without assignment returns a Series)

```
In [14]: newseries = df.loc[(df['BBB'] > 25) | (df['CCC'] >= -40), 'AAA']; newseries;
```

... or (with assignment modifies the DataFrame.)

```
In [15]: df.loc[(df['BBB'] > 25) | (df['CCC'] >= 75), 'AAA'] = 0.1; df
Out[15]:
   AAA  BBB  CCC
0  0.1   10  100
```

(continues on next page)

(continued from previous page)

```
1  5.0   20   50
2  0.1   30  -30
3  0.1   40  -50
```

Select rows with data closest to certain value using argsort

```
In [16]: df = pd.DataFrame(
.....:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]})
.....:
Out[16]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
2     6   30  -30
3     7   40  -50

In [17]: aValue = 43.0

In [18]: df.loc[(df.CCC-aValue).abs().argsort()]
Out[18]:
   AAA  BBB  CCC
1     5   20   50
0     4   10  100
2     6   30  -30
3     7   40  -50
```

Dynamically reduce a list of criteria using a binary operators

```
In [19]: df = pd.DataFrame(
.....:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]})
.....:
Out[19]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
2     6   30  -30
3     7   40  -50

In [20]: Crit1 = df.AAA <= 5.5

In [21]: Crit2 = df.BBB == 10.0

In [22]: Crit3 = df.CCC > -40.0
```

One could hard code:

```
In [23]: AllCrit = Crit1 & Crit2 & Crit3
```

...Or it can be done with a list of dynamically built criteria

```
In [24]: CritList = [Crit1,Crit2,Crit3]

In [25]: AllCrit = functools.reduce(lambda x,y: x & y, CritList)

In [26]: df[AllCrit]
Out[26]:
```

(continues on next page)

(continued from previous page)

|   | AAA | BBB | CCC |
|---|-----|-----|-----|
| 0 | 4   | 10  | 100 |

## 7.2 Selection

### 7.2.1 DataFrames

The *indexing* docs.

Using both row labels and value conditionals

```
In [27]: df = pd.DataFrame(
.....:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]})
.....:
```

```
Out [27]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
2     6   30  -30
3     7   40  -50
```

```
In [28]: df[(df.AAA <= 6) & (df.index.isin([0,2,4]))]
```

```
Out [28]:
```

```
↪
   AAA  BBB  CCC
0     4   10  100
2     6   30  -30
```

Use `loc` for label-oriented slicing and `iloc` positional slicing

```
In [29]: data = {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]}
```

```
In [30]: df = pd.DataFrame(data=data, index=['foo', 'bar', 'boo', 'kar'])
df
```

```
Out [30]:
   AAA  BBB  CCC
foo     4   10  100
bar     5   20   50
boo     6   30  -30
kar     7   40  -50
```

There are 2 explicit slicing methods, with a third general case

1. Positional-oriented (Python slicing style : exclusive of end)
2. Label-oriented (Non-Python slicing style : inclusive of end)
3. General (Either slicing style : depends on if the slice contains labels or positions)

```
In [31]: df.loc['bar':'kar'] #Label
```

```
Out [31]:
   AAA  BBB  CCC
bar     5   20   50
boo     6   30  -30
kar     7   40  -50
```

(continues on next page)

(continued from previous page)

```
# Generic
In [32]: df.iloc[0:3]
Out[32]:
   AAA  BBB  CCC
foo   4   10  100
bar   5   20   50
boo   6   30  -30

In [33]: df.loc['bar':'kar']
Out[33]:
   AAA  BBB  CCC
bar   5   20   50
boo   6   30  -30
kar   7   40  -50
```

Ambiguity arises when an index consists of integers with a non-zero start or non-unit increment.

```
In [34]: df2 = pd.DataFrame(data=data, index=[1,2,3,4]); #Note index starts at 1.

In [35]: df2.iloc[1:3] #Position-oriented
Out[35]:
   AAA  BBB  CCC
2     5   20   50
3     6   30  -30

In [36]: df2.loc[1:3] #Label-oriented
Out[36]:
   AAA  BBB  CCC
1     4   10  100
2     5   20   50
3     6   30  -30
```

Using inverse operator (~) to take the complement of a mask

```
In [37]: df = pd.DataFrame(
.....:     {'AAA' : [4,5,6,7], 'BBB' : [10,20,30,40], 'CCC' : [100,50,-30,-50]})
Out[37]:
   AAA  BBB  CCC
0     4   10  100
1     5   20   50
2     6   30  -30
3     7   40  -50

In [38]: df[~((df.AAA <= 6) & (df.index.isin([0,2,4])))]
Out[38]:
   AAA  BBB  CCC
1     5   20   50
3     7   40  -50
```



## 7.2.2 Panels

Extend a panel frame by transposing, adding a new dimension, and transposing back to the original dimensions

```
In [39]: rng = pd.date_range('1/1/2013', periods=100, freq='D')

In [40]: data = np.random.randn(100, 4)

In [41]: cols = ['A', 'B', 'C', 'D']

In [42]: df1, df2, df3 = pd.DataFrame(data, rng, cols), pd.DataFrame(data, rng, cols),
↳ pd.DataFrame(data, rng, cols)

In [43]: pf = pd.Panel({'df1':df1, 'df2':df2, 'df3':df3}); pf
Out[43]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 4 (minor_axis)
Items axis: df1 to df3
Major_axis axis: 2013-01-01 00:00:00 to 2013-04-10 00:00:00
Minor_axis axis: A to D

In [44]: pf.loc[:, :, 'F'] = pd.DataFrame(data, rng, cols); pf
////////////////////////////////////
↳
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 100 (major_axis) x 5 (minor_axis)
Items axis: df1 to df3
Major_axis axis: 2013-01-01 00:00:00 to 2013-04-10 00:00:00
Minor_axis axis: A to F
```

Mask a panel by using `np.where` and then reconstructing the panel with the new masked values

## 7.2.3 New Columns

Efficiently and dynamically creating new columns using `applymap`

```
In [45]: df = pd.DataFrame(
.....:     {'AAA' : [1,2,1,3], 'BBB' : [1,1,2,2], 'CCC' : [2,1,3,1]})
.....:
Out[45]:
   AAA  BBB  CCC
0    1    1    2
1    2    1    1
2    1    2    3
3    3    2    1

In [46]: source_cols = df.columns # or some subset would work too.

In [47]: new_cols = [str(x) + "_cat" for x in source_cols]

In [48]: categories = {1 : 'Alpha', 2 : 'Beta', 3 : 'Charlie' }

In [49]: df[new_cols] = df[source_cols].applymap(categories.get); df
Out[49]:
   AAA  BBB  CCC  AAA_cat BBB_cat  CCC_cat
0    1    1    2   Alpha   Alpha   Beta
```

(continues on next page)

(continued from previous page)

|   |   |   |   |         |       |         |
|---|---|---|---|---------|-------|---------|
| 1 | 2 | 1 | 1 | Beta    | Alpha | Alpha   |
| 2 | 1 | 2 | 3 | Alpha   | Beta  | Charlie |
| 3 | 3 | 2 | 1 | Charlie | Beta  | Alpha   |

Keep other columns when using min() with groupby

```
In [50]: df = pd.DataFrame(
.....:     {'AAA' : [1,1,1,2,2,2,3,3], 'BBB' : [2,1,3,4,5,1,2,3]})
.....:
Out[50]:
```

|   | AAA | BBB |
|---|-----|-----|
| 0 | 1   | 2   |
| 1 | 1   | 1   |
| 2 | 1   | 3   |
| 3 | 2   | 4   |
| 4 | 2   | 5   |
| 5 | 2   | 1   |
| 6 | 3   | 2   |
| 7 | 3   | 3   |

Method 1 : idxmin() to get the index of the mins

```
In [51]: df.loc[df.groupby("AAA")["BBB"].idxmin()]
Out[51]:
```

|   | AAA | BBB |
|---|-----|-----|
| 1 | 1   | 1   |
| 5 | 2   | 1   |
| 6 | 3   | 2   |

Method 2 : sort then take first of each

```
In [52]: df.sort_values(by="BBB").groupby("AAA", as_index=False).first()
Out[52]:
```

|   | AAA | BBB |
|---|-----|-----|
| 0 | 1   | 1   |
| 1 | 2   | 1   |
| 2 | 3   | 2   |

Notice the same results, with the exception of the index.

## 7.3 MultiIndexing

The *multindexing* docs.

Creating a multi-index from a labeled frame

```
In [53]: df = pd.DataFrame({'row' : [0,1,2],
.....:                     'One_X' : [1.1,1.1,1.1],
.....:                     'One_Y' : [1.2,1.2,1.2],
.....:                     'Two_X' : [1.11,1.11,1.11],
.....:                     'Two_Y' : [1.22,1.22,1.22]}); df
.....:
Out[53]:
```

|   | row | One_X | One_Y | Two_X | Two_Y |
|---|-----|-------|-------|-------|-------|
| 0 | 0   | 1.1   | 1.2   | 1.11  | 1.22  |

(continues on next page)

(continued from previous page)

```

1      1      1.1      1.2      1.11      1.22
2      2      1.1      1.2      1.11      1.22

# As Labelled Index
In [54]: df = df.set_index('row');df
////////////////////////////////////
↪
      One_X  One_Y  Two_X  Two_Y
row
0      1.1    1.2    1.11    1.22
1      1.1    1.2    1.11    1.22
2      1.1    1.2    1.11    1.22

# With Hierarchical Columns
In [55]: df.columns = pd.MultiIndex.from_tuples([tuple(c.split('_')) for c in df.
↪columns]);df
////////////////////////////////////
↪
      One      Two
      X      Y      X      Y
row
0      1.1  1.2  1.11  1.22
1      1.1  1.2  1.11  1.22
2      1.1  1.2  1.11  1.22

# Now stack & Reset
In [56]: df = df.stack(0).reset_index(1);df
////////////////////////////////////
↪
      level_1      X      Y
row
0      One  1.10  1.20
0      Two  1.11  1.22
1      One  1.10  1.20
1      Two  1.11  1.22
2      One  1.10  1.20
2      Two  1.11  1.22

# And fix the labels (Notice the label 'level_1' got added automatically)
In [57]: df.columns = ['Sample', 'All_X', 'All_Y'];df
////////////////////////////////////
↪
      Sample  All_X  All_Y
row
0      One  1.10  1.20
0      Two  1.11  1.22
1      One  1.10  1.20
1      Two  1.11  1.22
2      One  1.10  1.20
2      Two  1.11  1.22

```

### 7.3.1 Arithmetic

Performing arithmetic with a multi-index that needs broadcasting

```
In [58]: cols = pd.MultiIndex.from_tuples([(x,y) for x in ['A','B','C'] for y in ['O',
↳ 'I']])
```

```
In [59]: df = pd.DataFrame(np.random.randn(2,6),index=['n','m'],columns=cols); df
```

```
Out[59]:
```

|   | A         |           | B         |          | C        |           |
|---|-----------|-----------|-----------|----------|----------|-----------|
|   | O         | I         | O         | I        | O        | I         |
| n | 1.920906  | -0.388231 | -2.314394 | 0.665508 | 0.402562 | 0.399555  |
| m | -1.765956 | 0.850423  | 0.388054  | 0.992312 | 0.744086 | -0.739776 |

```
In [60]: df = df.div(df['C'],level=1); df
```

```
////////////////////////////////////
```

```
↳
```

|   | A         |           | B         |           | C   |     |
|---|-----------|-----------|-----------|-----------|-----|-----|
|   | O         | I         | O         | I         | O   | I   |
| n | 4.771702  | -0.971660 | -5.749162 | 1.665625  | 1.0 | 1.0 |
| m | -2.373321 | -1.149568 | 0.521518  | -1.341367 | 1.0 | 1.0 |

## 7.3.2 Slicing

Slicing a multi-index with xs

```
In [61]: coords = [('AA','one'),('AA','six'),('BB','one'),('BB','two'),('BB','six')]
```

```
In [62]: index = pd.MultiIndex.from_tuples(coords)
```

```
In [63]: df = pd.DataFrame([11,22,33,44,55],index,['MyData']); df
```

```
Out[63]:
```

|        | MyData |
|--------|--------|
| AA one | 11     |
| AA six | 22     |
| BB one | 33     |
| BB two | 44     |
| BB six | 55     |

To take the cross section of the 1st level and 1st axis the index:

```
In [64]: df.xs('BB',level=0,axis=0) #Note : level and axis are optional, and default_
↳ to zero
```

```
Out[64]:
```

|     | MyData |
|-----|--------|
| one | 33     |
| two | 44     |
| six | 55     |

... and now the 2nd level of the 1st axis.

```
In [65]: df.xs('six',level=1,axis=0)
```

```
Out[65]:
```

|    | MyData |
|----|--------|
| AA | 22     |
| BB | 55     |

Slicing a multi-index with xs, method #2



(continued from previous page)

```

////////////////////////////////////
↪
Student Course      I   II
Ada      Math      71  73
Quinn    Math      74  76
Violet   Math      77  79

In [77]: df.loc[(All, 'Math'), (All, 'II')]
////////////////////////////////////
↪
      Exams Labs
      II   II
Student Course
Ada      Math      73  74
Quinn    Math      76  77
Violet   Math      79  80

```

Setting portions of a multi-index with xs

### 7.3.3 Sorting

Sort by specific column or an ordered list of columns, with a multi-index

```

In [78]: df.sort_values(by=('Labs', 'II'), ascending=False)
Out[78]:

```

```

      Exams      Labs
      I   II      I   II
Student Course
Violet  Sci      78  81      81  81
        Math      77  79      81  80
        Comp      76  77      78  79
Quinn   Sci      75  78      78  78
        Math      74  76      78  77
        Comp      73  74      75  76
Ada     Sci      72  75      75  75
        Math      71  73      75  74
        Comp      70  71      72  73

```

Partial Selection, the need for sortedness;

### 7.3.4 Levels

Prepending a level to a multiindex

Flatten Hierarchical columns

## 7.4 Missing Data

The *missing data* docs.

Fill forward a reversed timeseries

```
In [79]: df = pd.DataFrame(np.random.randn(6,1), index=pd.date_range('2013-08-01',
↳ periods=6, freq='B'), columns=list('A'))
```

```
In [80]: df.loc[df.index[3], 'A'] = np.nan
```

```
In [81]: df
```

```
Out [81]:
```

```
          A
2013-08-01 -1.054874
2013-08-02 -0.179642
2013-08-05  0.639589
2013-08-06      NaN
2013-08-07  1.906684
2013-08-08  0.104050
```

```
In [82]: df.reindex(df.index[::-1]).ffill()
```

```
////////////////////////////////////
↳
          A
2013-08-08  0.104050
2013-08-07  1.906684
2013-08-06  1.906684
2013-08-05  0.639589
2013-08-02 -0.179642
2013-08-01 -1.054874
```

cumsum reset at NaN values

## 7.4.1 Replace

Using replace with backrefs

## 7.5 Grouping

The *grouping* docs.

Basic grouping with apply

Unlike agg, apply's callable is passed a sub-DataFrame which gives you access to all the columns

```
In [83]: df = pd.DataFrame({'animal': 'cat dog cat fish dog cat cat'.split(),
.....:                      'size': list('SSMMLL'),
.....:                      'weight': [8, 10, 11, 1, 20, 12, 12],
.....:                      'adult' : [False] * 5 + [True] * 2}); df
```

```
Out [83]:
```

```
  animal size  weight  adult
0   cat    S      8   False
1   dog    S     10   False
2   cat    M     11   False
3  fish    M      1   False
4   dog    M     20   False
5   cat    L     12    True
6   cat    L     12    True
```

(continues on next page)

(continued from previous page)

```
#List the size of the animals with the highest weight.
In [84]: df.groupby('animal').apply(lambda subf: subf['size'][subf['weight'].
↳idxmax()])
↳
animal
cat      L
dog      M
fish     M
dtype: object
```

### Using get\_group

```
In [85]: gb = df.groupby(['animal'])

In [86]: gb.get_group('cat')
Out[86]:
  animal size  weight  adult
0    cat   S        8   False
2    cat   M       11   False
5    cat   L       12    True
6    cat   L       12    True
```

### Apply to different items in a group

```
In [87]: def GrowUp(x):
.....:     avg_weight = sum(x[x['size'] == 'S'].weight * 1.5)
.....:     avg_weight += sum(x[x['size'] == 'M'].weight * 1.25)
.....:     avg_weight += sum(x[x['size'] == 'L'].weight)
.....:     avg_weight /= len(x)
.....:     return pd.Series(['L', avg_weight, True], index=['size', 'weight', 'adult'])
.....:

In [88]: expected_df = gb.apply(GrowUp)

In [89]: expected_df
Out[89]:
      size  weight  adult
animal
cat      L  12.4375   True
dog      L  20.0000   True
fish     L   1.2500   True
```

### Expanding Apply

```
In [90]: S = pd.Series([i / 100.0 for i in range(1,11)])

In [91]: def CumRet(x,y):
.....:     return x * (1 + y)
.....:

In [92]: def Red(x):
.....:     return functools.reduce(CumRet, x, 1.0)
.....:

In [93]: S.expanding().apply(Red, raw=True)
Out[93]:
```

(continues on next page)



(continued from previous page)

```

0    1.010000
1    1.030200
2    1.061106
3    1.103550
4    1.158728
5    1.228251
6    1.314229
7    1.419367
8    1.547110
9    1.701821
dtype: float64

```

### Replacing some values with mean of the rest of a group

```

In [94]: df = pd.DataFrame({'A' : [1, 1, 2, 2], 'B' : [1, -1, 1, 2]})

In [95]: gb = df.groupby('A')

In [96]: def replace(g):
....:     mask = g < 0
....:     g.loc[mask] = g[~mask].mean()
....:     return g
....:

In [97]: gb.transform(replace)
Out[97]:
   B
0  1.0
1  1.0
2  1.0
3  2.0

```

### Sort groups by aggregated data

```

In [98]: df = pd.DataFrame({'code': ['foo', 'bar', 'baz'] * 2,
....:                      'data': [0.16, -0.21, 0.33, 0.45, -0.59, 0.62],
....:                      'flag': [False, True] * 3})
....:

In [99]: code_groups = df.groupby('code')

In [100]: agg_n_sort_order = code_groups[['data']].transform(sum).sort_values(by='data
↪')

In [101]: sorted_df = df.loc[agg_n_sort_order.index]

In [102]: sorted_df
Out[102]:
   code  data  flag
1  bar -0.21  True
4  bar -0.59 False
0  foo  0.16 False
3  foo  0.45  True
2  baz  0.33 False
5  baz  0.62  True

```

### Create multiple aggregated columns

```
In [103]: rng = pd.date_range(start="2014-10-07", periods=10, freq='2min')
```

```
In [104]: ts = pd.Series(data = list(range(10)), index = rng)
```

```
In [105]: def MyCust(x):
.....:     if len(x) > 2:
.....:         return x[1] * 1.234
.....:     return pd.NaT
.....:
```

```
In [106]: mhc = {'Mean' : np.mean, 'Max' : np.max, 'Custom' : MyCust}
```

```
In [107]: ts.resample("5min").apply(mhc)
```

Out [107] :

|        |                     |       |
|--------|---------------------|-------|
| Custom | 2014-10-07 00:00:00 | 1.234 |
|        | 2014-10-07 00:05:00 | NaT   |
|        | 2014-10-07 00:10:00 | 7.404 |
|        | 2014-10-07 00:15:00 | NaT   |
| Max    | 2014-10-07 00:00:00 | 2     |
|        | 2014-10-07 00:05:00 | 4     |
|        | 2014-10-07 00:10:00 | 7     |
|        | 2014-10-07 00:15:00 | 9     |
| Mean   | 2014-10-07 00:00:00 | 1     |
|        | 2014-10-07 00:05:00 | 3.5   |
|        | 2014-10-07 00:10:00 | 6     |
|        | 2014-10-07 00:15:00 | 8.5   |

```
dtype: object
```

```
In [108]: ts
```

```

↪
2014-10-07 00:00:00      0
2014-10-07 00:02:00      1
2014-10-07 00:04:00      2
2014-10-07 00:06:00      3
2014-10-07 00:08:00      4
2014-10-07 00:10:00      5
2014-10-07 00:12:00      6
2014-10-07 00:14:00      7
2014-10-07 00:16:00      8
2014-10-07 00:18:00      9
Freq: 2T, dtype: int64

```

### Create a value counts column and reassign back to the DataFrame

```
In [109]: df = pd.DataFrame({'Color': 'Red Red Red Blue'.split(),
.....:                       'Value': [100, 150, 50, 50]}); df
```

Out [109]:

|   | Color | Value |
|---|-------|-------|
| 0 | Red   | 100   |
| 1 | Red   | 150   |
| 2 | Red   | 50    |
| 3 | Blue  | 50    |

```
In [110]: df['Counts'] = df.groupby(['Color']).transform(len)
```

(continues on next page)

(continued from previous page)

```
In [111]: df
Out[111]:
```

|   | Color | Value | Counts |
|---|-------|-------|--------|
| 0 | Red   | 100   | 3      |
| 1 | Red   | 150   | 3      |
| 2 | Red   | 50    | 3      |
| 3 | Blue  | 50    | 1      |

Shift groups of the values in a column based on the index

```
In [112]: df = pd.DataFrame(
.....:     {'line_race': [10, 10, 8, 10, 10, 8],
.....:      'beyer': [99, 102, 103, 103, 88, 100]},
.....:     index=[u'Last Gunfighter', u'Last Gunfighter', u'Last Gunfighter',
.....:            u'Paynter', u'Paynter', u'Paynter']); df
Out[112]:
```

|                 | line_race | beyer |
|-----------------|-----------|-------|
| Last Gunfighter | 10        | 99    |
| Last Gunfighter | 10        | 102   |
| Last Gunfighter | 8         | 103   |
| Paynter         | 10        | 103   |
| Paynter         | 10        | 88    |
| Paynter         | 8         | 100   |

```
In [113]: df['beyer_shifted'] = df.groupby(level=0)['beyer'].shift(1)

In [114]: df
Out[114]:
```

|                 | line_race | beyer | beyer_shifted |
|-----------------|-----------|-------|---------------|
| Last Gunfighter | 10        | 99    | NaN           |
| Last Gunfighter | 10        | 102   | 99.0          |
| Last Gunfighter | 8         | 103   | 102.0         |
| Paynter         | 10        | 103   | NaN           |
| Paynter         | 10        | 88    | 103.0         |
| Paynter         | 8         | 100   | 88.0          |

Select row with maximum value from each group

```
In [115]: df = pd.DataFrame({'host': ['other', 'other', 'that', 'this', 'this'],
.....:                       'service': ['mail', 'web', 'mail', 'mail', 'web'],
.....:                       'no': [1, 2, 1, 2, 1]}).set_index(['host', 'service'])
.....:

In [116]: mask = df.groupby(level=0).agg('idxmax')

In [117]: df_count = df.loc[mask['no']].reset_index()

In [118]: df_count
Out[118]:
```

|   | host  | service | no |
|---|-------|---------|----|
| 0 | other | web     | 2  |
| 1 | that  | mail    | 1  |
| 2 | this  | mail    | 2  |

Grouping like Python's `itertools.groupby`

```
In [119]: df = pd.DataFrame([0, 1, 0, 1, 1, 1, 0, 1, 1], columns=['A'])
```

```
In [120]: df.A.groupby((df.A != df.A.shift()).cumsum()).groups
```

```
Out[120]:
```

```
{1: Int64Index([0], dtype='int64'),
 2: Int64Index([1], dtype='int64'),
 3: Int64Index([2], dtype='int64'),
 4: Int64Index([3, 4, 5], dtype='int64'),
 5: Int64Index([6], dtype='int64'),
 6: Int64Index([7, 8], dtype='int64')}
```

```
In [121]: df.A.groupby((df.A != df.A.shift()).cumsum()).cumsum()
```

```

0      0
1      1
2      0
3      1
4      2
5      3
6      0
7      1
8      2
Name: A, dtype: int64
```

## 7.5.1 Expanding Data

Alignment and to-date

Rolling Computation window based on values instead of counts

Rolling Mean by Time Interval

## 7.5.2 Splitting

Splitting a frame

Create a list of dataframes, split using a delineation based on logic included in rows.

```
In [122]: df = pd.DataFrame(data={'Case' : ['A', 'A', 'A', 'B', 'A', 'A', 'B', 'A', 'A'],
.....:                           'Data' : np.random.randn(9)})
.....:
```

```
In [123]: dfs = list(zip(*df.groupby((1*(df['Case']=='B')).cumsum().rolling(window=3,
↳min_periods=1).median())))[-1])
```

```
In [124]: dfs[0]
```

```
Out[124]:
```

```
Case      Data
0      A  0.174068
1      A -0.439461
2      A -0.741343
3      B -0.079673
```

```
In [125]: dfs[1]
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////Out [125]:
↪
Case      Data
4      A -0.922875
5      A  0.303638
6      B -0.917368

In [126]: dfs[2]
////////////////////////////////////
↪
Case      Data
7      A -1.624062
8      A -0.758514

```

### 7.5.3 Pivot

The *Pivot* docs.

Partial sums and subtotals

```

In [127]: df = pd.DataFrame(data={'Province' : ['ON','QC','BC','AL','AL','MN','ON'],
.....:                               'City' : ['Toronto','Montreal','Vancouver','Calgary',
↪      'Edmonton','Winnipeg','Windsor'],
.....:                               'Sales' : [13,6,16,8,4,3,1]})
.....:

In [128]: table = pd.pivot_table(df, values='Sales', index=['Province'], columns=['City',
↪      ''], aggfunc=np.sum, margins=True)

In [129]: table.stack('City')
Out[129]:

```

| Province | City      | Sales |
|----------|-----------|-------|
| AL       | All       | 12.0  |
|          | Calgary   | 8.0   |
|          | Edmonton  | 4.0   |
| BC       | All       | 16.0  |
|          | Vancouver | 16.0  |
| MN       | All       | 3.0   |
|          | Winnipeg  | 3.0   |
| ...      |           | ...   |
| All      | Calgary   | 8.0   |
|          | Edmonton  | 4.0   |
|          | Montreal  | 6.0   |
|          | Toronto   | 13.0  |
|          | Vancouver | 16.0  |
|          | Windsor   | 1.0   |
|          | Winnipeg  | 3.0   |

[20 rows x 1 columns]

Frequency table like *plyr* in R

```

In [130]: grades = [48,99,75,80,42,80,72,68,36,78]

```

(continues on next page)

(continued from previous page)

```

In [131]: df = pd.DataFrame( {'ID': ["x%d" % r for r in range(10)],
.....:                      'Gender' : ['F', 'M', 'F', 'M', 'F', 'M', 'F', 'M', 'M',
↪ 'M'],
.....:                      'ExamYear': ['2007','2007','2007','2008','2008','2008',
↪ '2008','2009','2009','2009'],
.....:                      'Class': ['algebra', 'stats', 'bio', 'algebra', 'algebra
↪ ', 'stats', 'stats', 'algebra', 'bio', 'bio'],
.....:                      'Participated': ['yes','yes','yes','yes','no','yes','yes
↪ ','yes','yes','yes'],
.....:                      'Passed': ['yes' if x > 50 else 'no' for x in grades],
.....:                      'Employed': [True,True,True,False,False,False,False,
↪ True,True,False],
.....:                      'Grade': grades})

In [132]: df.groupby('ExamYear').agg({'Participated': lambda x: x.value_counts()['yes
↪ '],
.....:                      'Passed': lambda x: sum(x == 'yes'),
.....:                      'Employed' : lambda x : sum(x),
.....:                      'Grade' : lambda x : sum(x) / len(x)})

Out[132]:

```

|          | Participated | Passed | Employed | Grade     |
|----------|--------------|--------|----------|-----------|
| ExamYear |              |        |          |           |
| 2007     | 3            | 2      | 3        | 74.000000 |
| 2008     | 3            | 3      | 0        | 68.500000 |
| 2009     | 3            | 2      | 2        | 60.666667 |

### Plot pandas DataFrame with year over year data

To create year and month crosstabulation:

```

In [133]: df = pd.DataFrame({'value': np.random.randn(36)},
.....:                      index=pd.date_range('2011-01-01', freq='M', periods=36))
.....:

In [134]: pd.pivot_table(df, index=df.index.month, columns=df.index.year,
.....:                      values='value', aggfunc='sum')
.....:

Out[134]:

```

|    | 2011      | 2012      | 2013      |
|----|-----------|-----------|-----------|
| 1  | -0.560859 | 0.120930  | 0.516870  |
| 2  | -0.589005 | -0.210518 | 0.343125  |
| 3  | -1.070678 | -0.931184 | 2.137827  |
| 4  | -1.681101 | 0.240647  | 0.452429  |
| 5  | 0.403776  | -0.027462 | 0.483103  |
| 6  | 0.609862  | 0.033113  | 0.061495  |
| 7  | 0.387936  | -0.658418 | 0.240767  |
| 8  | 1.815066  | 0.324102  | 0.782413  |
| 9  | 0.705200  | -1.403048 | 0.628462  |
| 10 | -0.668049 | -0.581967 | -0.880627 |
| 11 | 0.242501  | -1.233862 | 0.777575  |
| 12 | 0.313421  | -3.520876 | -0.779367 |

## 7.5.4 Apply

Rolling Apply to Organize - Turning embedded lists into a multi-index frame

```
In [135]: df = pd.DataFrame(data={'A' : [[2,4,8,16],[100,200],[10,20,30]], 'B' : [['a'
→ 'b','c'],['jj','kk'],['ccc']]},index=['I','II','III'])

In [136]: def SeriesFromSubList(aList):
.....:     return pd.Series(aList)
.....:

In [137]: df_orgz = pd.concat(dict([ (ind,row.apply(SeriesFromSubList)) for ind,row_
→ in df.iterrows() ]))
```

Rolling Apply with a DataFrame returning a Series

Rolling Apply to multiple columns where function calculates a Series before a Scalar from the Series is returned

```
In [138]: df = pd.DataFrame(data=np.random.randn(2000,2)/10000,
.....:                      index=pd.date_range('2001-01-01',periods=2000),
.....:                      columns=['A','B']); df
.....:

Out[138]:
```

|            | A         | B         |
|------------|-----------|-----------|
| 2001-01-01 | 0.000032  | -0.000004 |
| 2001-01-02 | -0.000001 | 0.000207  |
| 2001-01-03 | 0.000120  | -0.000220 |
| 2001-01-04 | -0.000083 | -0.000165 |
| 2001-01-05 | -0.000047 | 0.000156  |
| 2001-01-06 | 0.000027  | 0.000104  |
| 2001-01-07 | 0.000041  | -0.000101 |
| ...        | ...       | ...       |
| 2006-06-17 | -0.000034 | 0.000034  |
| 2006-06-18 | 0.000002  | 0.000166  |
| 2006-06-19 | 0.000023  | -0.000081 |
| 2006-06-20 | -0.000061 | 0.000012  |
| 2006-06-21 | -0.000111 | 0.000027  |
| 2006-06-22 | -0.000061 | -0.000009 |
| 2006-06-23 | 0.000074  | -0.000138 |

```
[2000 rows x 2 columns]

In [139]: def gm(aDF,Const):
.....:     v = (((aDF.A+aDF.B)+1).cumprod()-1)*Const
.....:     return (aDF.index[0],v.iloc[-1])
.....:

In [140]: S = pd.Series(dict([ gm(df.iloc[i:min(i+51,len(df)-1)],5) for i in_
→ range(len(df)-50) ])); S

Out[140]:
```

|            |           |
|------------|-----------|
| 2001-01-01 | -0.001373 |
| 2001-01-02 | -0.001705 |
| 2001-01-03 | -0.002885 |
| 2001-01-04 | -0.002987 |
| 2001-01-05 | -0.002384 |
| 2001-01-06 | -0.004700 |
| 2001-01-07 | -0.005500 |
| ...        | ...       |

(continues on next page)

(continued from previous page)

```

2006-04-28    -0.002682
2006-04-29    -0.002436
2006-04-30    -0.002602
2006-05-01    -0.001785
2006-05-02    -0.001799
2006-05-03    -0.000605
2006-05-04    -0.000541
Length: 1950, dtype: float64

```

### Rolling apply with a DataFrame returning a Scalar

Rolling Apply to multiple columns where function returns a Scalar (Volume Weighted Average Price)

```

In [141]: rng = pd.date_range(start = '2014-01-01', periods = 100)

In [142]: df = pd.DataFrame({'Open' : np.random.randn(len(rng)),
.....:                      'Close' : np.random.randn(len(rng)),
.....:                      'Volume' : np.random.randint(100, 2000, len(rng))},
➔index=rng); df
.....:
Out[142]:
           Open      Close  Volume
2014-01-01  0.011174 -0.653039   1581
2014-01-02  0.214258  1.314205   1707
2014-01-03 -1.046922 -0.341915   1768
2014-01-04 -0.752902 -1.303586    836
2014-01-05 -0.410793  0.396288    694
2014-01-06  0.648401 -0.548006    796
2014-01-07  0.737320  0.481380    265
...          ...      ...      ...
2014-04-04  0.120378 -2.548128    564
2014-04-05  0.231661  0.223346   1908
2014-04-06  0.952664  1.228841   1090
2014-04-07 -0.176090  0.552784   1813
2014-04-08  1.781318 -0.795389   1103
2014-04-09 -0.753493 -0.018815   1456
2014-04-10 -1.047997  1.138197   1193

[100 rows x 3 columns]

In [143]: def vwap(bars): return ((bars.Close*bars.Volume).sum()/bars.Volume.sum())

In [144]: window = 5

In [145]: s = pd.concat([ (pd.Series(vwap(df.iloc[i:i+window])), index=[df.
➔index[i+window]]) for i in range(len(df)-window) ]);

In [146]: s.round(2)
Out[146]:
2014-01-06    -0.03
2014-01-07     0.07
2014-01-08   -0.40
2014-01-09   -0.81
2014-01-10   -0.63
2014-01-11   -0.86
2014-01-12   -0.36
...

```

(continues on next page)



(continued from previous page)

```

2014-04-04    -1.27
2014-04-05    -1.36
2014-04-06    -0.73
2014-04-07     0.04
2014-04-08     0.21
2014-04-09     0.07
2014-04-10     0.25
Length: 95, dtype: float64

```

## 7.6 Timeseries

Between times

Using indexer between time

Constructing a datetime range that excludes weekends and includes only certain times

Vectorized Lookup

Aggregation and plotting time series

Turn a matrix with hours in columns and days in rows into a continuous row sequence in the form of a time series.

How to rearrange a Python pandas DataFrame?

Dealing with duplicates when reindexing a timeseries to a specified frequency

Calculate the first day of the month for each entry in a DatetimeIndex

```

In [147]: dates = pd.date_range('2000-01-01', periods=5)

In [148]: dates.to_period(freq='M').to_timestamp()
Out[148]:
DatetimeIndex(['2000-01-01', '2000-01-01', '2000-01-01', '2000-01-01',
                '2000-01-01'],
              dtype='datetime64[ns]', freq=None)

```

### 7.6.1 Resampling

The *Resample* docs.

Using Grouper instead of TimeGrouper for time grouping of values

Time grouping with some missing values

Valid frequency arguments to Grouper

Grouping using a MultiIndex

Using TimeGrouper and another grouping to create subgroups, then apply a custom function

Resampling with custom periods

Resample intraday frame without adding new days

Resample minute data

Resample with groupby

## 7.7 Merge

The *Concat* docs. The *Join* docs.

Append two dataframes with overlapping index (emulate R rbind)

```
In [149]: rng = pd.date_range('2000-01-01', periods=6)

In [150]: df1 = pd.DataFrame(np.random.randn(6, 3), index=rng, columns=['A', 'B', 'C']
↳ )

In [151]: df2 = df1.copy()
```

Depending on df construction, ignore\_index may be needed

```
In [152]: df = df1.append(df2, ignore_index=True); df
Out[152]:
```

|    | A         | B         | C         |
|----|-----------|-----------|-----------|
| 0  | -0.480676 | -1.305282 | -0.212846 |
| 1  | 1.979901  | 0.363112  | -0.275732 |
| 2  | -1.433852 | 0.580237  | -0.013672 |
| 3  | 1.776623  | -0.803467 | 0.521517  |
| 4  | -0.302508 | -0.442948 | -0.395768 |
| 5  | -0.249024 | -0.031510 | 2.413751  |
| 6  | -0.480676 | -1.305282 | -0.212846 |
| 7  | 1.979901  | 0.363112  | -0.275732 |
| 8  | -1.433852 | 0.580237  | -0.013672 |
| 9  | 1.776623  | -0.803467 | 0.521517  |
| 10 | -0.302508 | -0.442948 | -0.395768 |
| 11 | -0.249024 | -0.031510 | 2.413751  |

Self Join of a DataFrame

```
In [153]: df = pd.DataFrame(data={'Area' : ['A'] * 5 + ['C'] * 2,
.....:                          'Bins' : [110] * 2 + [160] * 3 + [40] * 2,
.....:                          'Test_0' : [0, 1, 0, 1, 2, 0, 1],
.....:                          'Data' : np.random.randn(7)});df
Out[153]:
```

|   | Area | Bins | Test_0 | Data      |
|---|------|------|--------|-----------|
| 0 | A    | 110  | 0      | -0.378914 |
| 1 | A    | 110  | 1      | -1.032527 |
| 2 | A    | 160  | 0      | -1.402816 |
| 3 | A    | 160  | 1      | 0.715333  |
| 4 | A    | 160  | 2      | -0.091438 |
| 5 | C    | 40   | 0      | 1.608418  |
| 6 | C    | 40   | 1      | 0.753207  |

```
In [154]: df['Test_1'] = df['Test_0'] - 1

In [155]: pd.merge(df, df, left_on=['Bins', 'Area', 'Test_0'], right_on=['Bins', 'Area']
↳ , 'Test_1', suffixes=('_L', '_R'))
Out[155]:
```

|   | Area | Bins | Test_0_L | Data_L    | Test_1_L | Test_0_R | Data_R    | Test_1_R |
|---|------|------|----------|-----------|----------|----------|-----------|----------|
| 0 | A    | 110  | 0        | -0.378914 | -1       | 1        | -1.032527 | 0        |
| 1 | A    | 160  | 0        | -1.402816 | -1       | 1        | 0.715333  | 0        |
| 2 | A    | 160  | 1        | 0.715333  | 0        | 2        | -0.091438 | 1        |
| 3 | C    | 40   | 0        | 1.608418  | -1       | 1        | 0.753207  | 0        |

How to set the index and join

KDB like asof join

Join with a criteria based on the values

Using searchsorted to merge based on values inside a range

## 7.8 Plotting

The *Plotting* docs.

Make Matplotlib look like R

Setting x-axis major and minor labels

Plotting multiple charts in an ipython notebook

Creating a multi-line plot

Plotting a heatmap

Annotate a time-series plot

Annotate a time-series plot #2

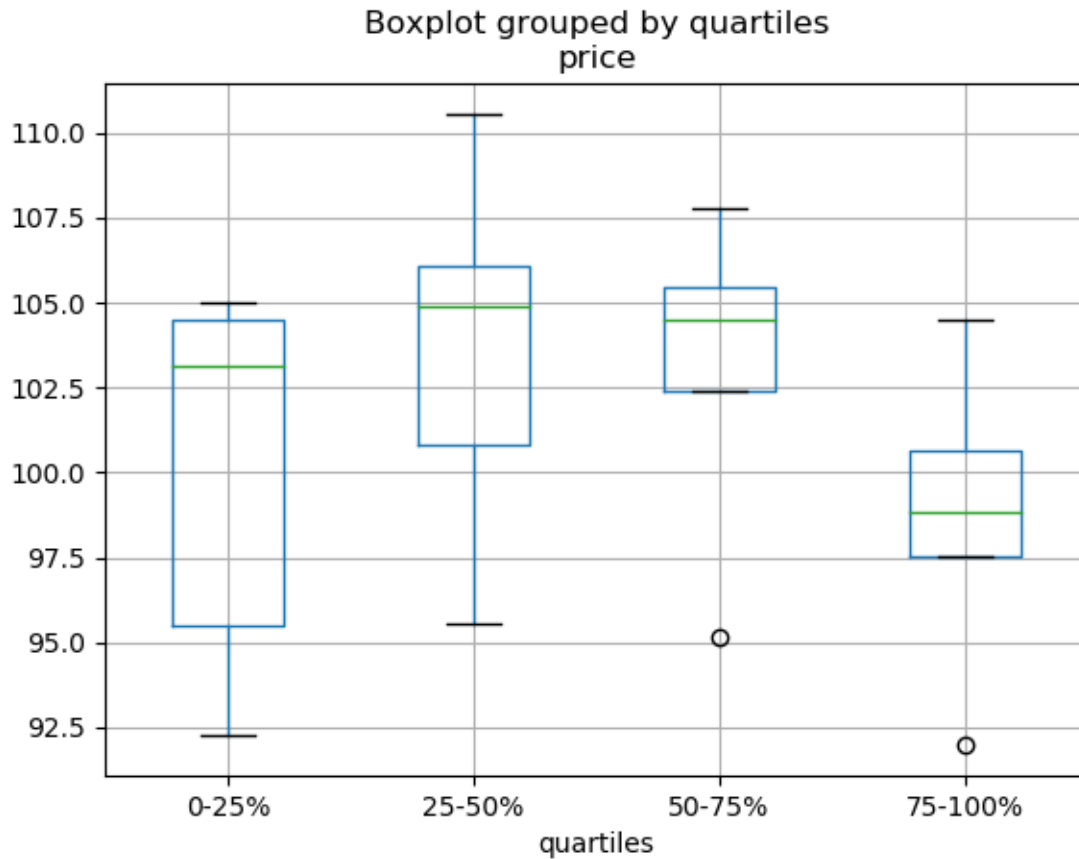
Generate Embedded plots in excel files using Pandas, Vincent and xlsxwriter

Boxplot for each quartile of a stratifying variable

```
In [156]: df = pd.DataFrame(
.....:     {u'stratifying_var': np.random.uniform(0, 100, 20),
.....:      u'price': np.random.normal(100, 5, 20)})
.....:

In [157]: df[u'quartiles'] = pd.qcut(
.....:     df[u'stratifying_var'],
.....:     4,
.....:     labels=[u'0-25%', u'25-50%', u'50-75%', u'75-100%'])
.....:

In [158]: df.boxplot(column=u'price', by=u'quartiles')
Out[158]: <matplotlib.axes._subplots.AxesSubplot at 0x11f845470>
```



## 7.9 Data In/Out

Performance comparison of SQL vs HDF5

### 7.9.1 CSV

The *CSV* docs

`read_csv` in action

appending to a csv

Reading a csv chunk-by-chunk

Reading only certain rows of a csv chunk-by-chunk

Reading the first few lines of a frame

Reading a file that is compressed but not by `gzip/bz2` (the native compressed formats which `read_csv` understands). This example shows a WinZipped file, but is a general application of opening the file within a context manager and using that handle to read. [See here](#)

Inferring dtypes from a file

Dealing with bad lines

Dealing with bad lines II

Reading CSV with Unix timestamps and converting to local timezone

Write a multi-row index CSV without writing duplicates

### 7.9.1.1 Reading multiple files to create a single DataFrame

The best way to combine multiple files into a single DataFrame is to read the individual frames one by one, put all of the individual frames into a list, and then combine the frames in the list using `pd.concat()`:

```
In [159]: for i in range(3):
.....:     data = pd.DataFrame(np.random.randn(10, 4))
.....:     data.to_csv('file_{}.csv'.format(i))
.....:

In [160]: files = ['file_0.csv', 'file_1.csv', 'file_2.csv']

In [161]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

You can use the same approach to read all files matching a pattern. Here is an example using `glob`:

```
In [162]: import glob

In [163]: files = glob.glob('file_*.csv')

In [164]: result = pd.concat([pd.read_csv(f) for f in files], ignore_index=True)
```

Finally, this strategy will work with the other `pd.read_*` (...) functions described in the *io docs*.

### 7.9.1.2 Parsing date components in multi-columns

Parsing date components in multi-columns is faster with a format

```
In [30]: i = pd.date_range('20000101', periods=10000)

In [31]: df = pd.DataFrame(dict(year = i.year, month = i.month, day = i.day))

In [32]: df.head()
Out[32]:
   day  month  year
0    1     1  2000
1    2     1  2000
2    3     1  2000
3    4     1  2000
4    5     1  2000

In [33]: %timeit pd.to_datetime(df.year*10000+df.month*100+df.day, format='%Y%m%d')
100 loops, best of 3: 7.08 ms per loop

# simulate combining into a string, then parsing
In [34]: ds = df.apply(lambda x: "%04d%02d%02d" % (x['year'], x['month'], x['day']),
    ↪ axis=1)

In [35]: ds.head()
Out[35]:
```

(continues on next page)

(continued from previous page)

```
0    20000101
1    20000102
2    20000103
3    20000104
4    20000105
dtype: object

In [36]: %timeit pd.to_datetime(ds)
1 loops, best of 3: 488 ms per loop
```

### 7.9.1.3 Skip row between header and data

```
In [165]: data = """;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   ;;;;
.....:   date;Param1;Param2;Param4;Param5
.....:       ;m2; °C;m2;m
.....:   ;;;;
.....:   01.01.1990  00:00;1;1;2;3
.....:   01.01.1990  01:00;5;3;4;5
.....:   01.01.1990  02:00;9;5;6;7
.....:   01.01.1990  03:00;13;7;8;9
.....:   01.01.1990  04:00;17;9;10;11
.....:   01.01.1990  05:00;21;11;12;13
.....:   """
.....:
.....:
```

### Option 1: pass rows explicitly to skiprows

```
In [166]: pd.read_csv(StringIO(data), sep=';', skiprows=[11,12],
.....:                  index_col=0, parse_dates=True, header=10)
.....:
Out[166]:
```

|                     | Param1 | Param2 | Param4 | Param5 |
|---------------------|--------|--------|--------|--------|
| date                |        |        |        |        |
| 1990-01-01 00:00:00 | 1      | 1      | 2      | 3      |
| 1990-01-01 01:00:00 | 5      | 3      | 4      | 5      |
| 1990-01-01 02:00:00 | 9      | 5      | 6      | 7      |
| 1990-01-01 03:00:00 | 13     | 7      | 8      | 9      |
| 1990-01-01 04:00:00 | 17     | 9      | 10     | 11     |
| 1990-01-01 05:00:00 | 21     | 11     | 12     | 13     |

## Option 2: read column names and then data

```
In [167]: pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns
Out[167]: Index(['date', 'Param1', 'Param2', 'Param4', 'Param5'], dtype='object')
```

```
In [168]: columns = pd.read_csv(StringIO(data), sep=';', header=10, nrows=10).columns
```

```
In [169]: pd.read_csv(StringIO(data), sep=';', index_col=0,
.....:                header=12, parse_dates=True, names=columns)
.....:
Out[169]:
```

|                     | Param1 | Param2 | Param4 | Param5 |
|---------------------|--------|--------|--------|--------|
| date                |        |        |        |        |
| 1990-01-01 00:00:00 | 1      | 1      | 2      | 3      |
| 1990-01-01 01:00:00 | 5      | 3      | 4      | 5      |
| 1990-01-01 02:00:00 | 9      | 5      | 6      | 7      |
| 1990-01-01 03:00:00 | 13     | 7      | 8      | 9      |
| 1990-01-01 04:00:00 | 17     | 9      | 10     | 11     |
| 1990-01-01 05:00:00 | 21     | 11     | 12     | 13     |

## 7.9.2 SQL

The *SQL* docs

Reading from databases with SQL

## 7.9.3 Excel

The *Excel* docs

Reading from a filelike handle

Modifying formatting in XlsxWriter output

## 7.9.4 HTML

Reading HTML tables from a server that cannot handle the default request header

## 7.9.5 HDFStore

The *HDFStores* docs

Simple Queries with a Timestamp Index

Managing heterogeneous data using a linked multiple table hierarchy

Merging on-disk tables with millions of rows

Avoiding inconsistencies when writing to a store from multiple processes/threads

De-duplicating a large store by chunks, essentially a recursive reduction operation. Shows a function for taking in data from csv file and creating a store by chunks, with date parsing as well. [See here](#)

Creating a store chunk-by-chunk from a csv file

Appending to a store, while creating a unique index

Large Data work flows

Reading in a sequence of files, then providing a global unique index to a store while appending

Groupby on a HDFStore with low group density

Groupby on a HDFStore with high group density

Hierarchical queries on a HDFStore

Counting with a HDFStore

Troubleshoot HDFStore exceptions

Setting min\_itemsize with strings

Using ptpack to create a completely-sorted-index on a store

Storing Attributes to a group node

```
In [170]: df = pd.DataFrame(np.random.randn(8,3))

In [171]: store = pd.HDFStore('test.h5')

In [172]: store.put('df',df)

# you can store an arbitrary Python object via pickle
In [173]: store.get_storer('df').attrs.my_attribute = dict(A = 10)

In [174]: store.get_storer('df').attrs.my_attribute
Out[174]: {'A': 10}
```

## 7.9.6 Binary Files

pandas readily accepts NumPy record arrays, if you need to read in a binary file consisting of an array of C structs. For example, given this C program in a file called `main.c` compiled with `gcc main.c -std=gnu99` on a 64-bit machine,

```
#include <stdio.h>
#include <stdint.h>

typedef struct _Data
{
    int32_t count;
    double avg;
    float scale;
} Data;

int main(int argc, const char *argv[])
{
    size_t n = 10;
    Data d[n];

    for (int i = 0; i < n; ++i)
    {
        d[i].count = i;
        d[i].avg = i + 1.0;
        d[i].scale = (float) i + 2.0f;
    }
}
```

(continues on next page)



(continued from previous page)

```

FILE *file = fopen("binary.dat", "wb");
fwrite(&d, sizeof(Data), n, file);
fclose(file);

return 0;
}

```

the following Python code will read the binary file 'binary.dat' into a pandas DataFrame, where each element of the struct corresponds to a column in the frame:

```

names = 'count', 'avg', 'scale'

# note that the offsets are larger than the size of the type because of
# struct padding
offsets = 0, 8, 16
formats = 'i4', 'f8', 'f4'
dt = np.dtype({'names': names, 'offsets': offsets, 'formats': formats},
              align=True)
df = pd.DataFrame(np.fromfile('binary.dat', dt))

```

**Note:** The offsets of the structure elements may be different depending on the architecture of the machine on which the file was created. Using a raw binary file format like this for general data storage is not recommended, as it is not cross platform. We recommended either HDF5 or msgpack, both of which are supported by pandas' IO facilities.

## 7.10 Computation

Numerical integration (sample-based) of a time series

### 7.11 Timedeltas

The *Timedeltas* docs.

Using *timedeltas*

```

In [175]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [176]: s - s.max()
Out[176]:
0    -2 days
1    -1 days
2     0 days
dtype: timedelta64[ns]

In [177]: s.max() - s
Out[177]:
0     2 days
1     1 days
2     0 days
dtype: timedelta64[ns]

```

(continues on next page)

(continued from previous page)

```
In [178]: s - datetime.datetime(2011,1,1,3,5)
```

```

////////////////////////////////////
↪
0    364 days 20:55:00
1    365 days 20:55:00
2    366 days 20:55:00
dtype: timedelta64[ns]
```

```
In [179]: s + datetime.timedelta(minutes=5)
```

```

////////////////////////////////////
↪
0    2012-01-01 00:05:00
1    2012-01-02 00:05:00
2    2012-01-03 00:05:00
dtype: datetime64[ns]
```

```
In [180]: datetime.datetime(2011,1,1,3,5) - s
```

```

////////////////////////////////////
↪
0    -365 days +03:05:00
1    -366 days +03:05:00
2    -367 days +03:05:00
dtype: timedelta64[ns]
```

```
In [181]: datetime.timedelta(minutes=5) + s
```

```

////////////////////////////////////
↪
0    2012-01-01 00:05:00
1    2012-01-02 00:05:00
2    2012-01-03 00:05:00
dtype: datetime64[ns]
```

### Adding and subtracting deltas and dates

```
In [182]: deltas = pd.Series([ datetime.timedelta(days=i) for i in range(3) ])
```

```
In [183]: df = pd.DataFrame(dict(A = s, B = deltas)); df
```

```
Out[183]:
```

|   | A                 | B |
|---|-------------------|---|
| 0 | 2012-01-01 0 days |   |
| 1 | 2012-01-02 1 days |   |
| 2 | 2012-01-03 2 days |   |

```
In [184]: df['New Dates'] = df['A'] + df['B'];
```

```
In [185]: df['Delta'] = df['A'] - df['New Dates']; df
```

```
Out[185]:
```

|   | A                 | B | New Dates          | Delta |
|---|-------------------|---|--------------------|-------|
| 0 | 2012-01-01 0 days |   | 2012-01-01 0 days  |       |
| 1 | 2012-01-02 1 days |   | 2012-01-03 -1 days |       |
| 2 | 2012-01-03 2 days |   | 2012-01-05 -2 days |       |

```
In [186]: df.dtypes
```

```

////////////////////////////////////
↪
A    datetime64[ns]
```

(continues on next page)

(continued from previous page)

```

B          timedelta64[ns]
New Dates   datetime64[ns]
Delta       timedelta64[ns]
dtype: object

```

### Another example

Values can be set to NaT using np.nan, similar to datetime

```

In [187]: y = s - s.shift(); y
Out[187]:
0      NaT
1    1 days
2    1 days
dtype: timedelta64[ns]

In [188]: y[1] = np.nan; y
Out[188]:
0      NaT
1      NaT
2    1 days
dtype: timedelta64[ns]

```

## 7.12 Aliasing Axis Names

To globally provide aliases for axis names, one can define these 2 functions:

```

In [189]: def set_axis_alias(cls, axis, alias):
.....:     if axis not in cls._AXIS_NUMBERS:
.....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
.....:     cls._AXIS_ALIASES[alias] = axis
.....:

```

```

In [190]: def clear_axis_alias(cls, axis, alias):
.....:     if axis not in cls._AXIS_NUMBERS:
.....:         raise Exception("invalid axis [%s] for alias [%s]" % (axis, alias))
.....:     cls._AXIS_ALIASES.pop(alias, None)
.....:

```

```

In [191]: set_axis_alias(pd.DataFrame, 'columns', 'myaxis2')

In [192]: df2 = pd.DataFrame(np.random.randn(3,2), columns=['c1', 'c2'], index=['i1', 'i2', 'i3'])

In [193]: df2.sum(axis='myaxis2')
Out[193]:
i1    0.745167
i2   -0.176251
i3    0.014354
dtype: float64

In [194]: clear_axis_alias(pd.DataFrame, 'columns', 'myaxis2')

```

## 7.13 Creating Example Data

To create a dataframe from every combination of some given values, like R's `expand.grid()` function, we can create a dict where the keys are column names and the values are lists of the data values:

```
In [195]: def expand_grid(data_dict):
.....:     rows = itertools.product(*data_dict.values())
.....:     return pd.DataFrame.from_records(rows, columns=data_dict.keys())
.....:

In [196]: df = expand_grid(
.....:     {'height': [60, 70],
.....:      'weight': [100, 140, 180],
.....:      'sex': ['Male', 'Female']})
.....:

In [197]: df
Out[197]:
```

|    | height | weight | sex    |
|----|--------|--------|--------|
| 0  | 60     | 100    | Male   |
| 1  | 60     | 100    | Female |
| 2  | 60     | 140    | Male   |
| 3  | 60     | 140    | Female |
| 4  | 60     | 180    | Male   |
| 5  | 60     | 180    | Female |
| 6  | 70     | 100    | Male   |
| 7  | 70     | 100    | Female |
| 8  | 70     | 140    | Male   |
| 9  | 70     | 140    | Female |
| 10 | 70     | 180    | Male   |
| 11 | 70     | 180    | Female |

## INTRO TO DATA STRUCTURES

We'll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, and axis labeling / alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

```
In [1]: import numpy as np
In [2]: import pandas as pd
```

Here is a basic tenet to keep in mind: **data alignment is intrinsic**. The link between labels and data will not be broken unless done so explicitly by you.

We'll give a brief intro to the data structures, then consider all of the broad categories of functionality and methods in separate sections.

### 8.1 Series

*Series* is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the **index**. The basic method to create a Series is to call:

```
>>> s = pd.Series(data, index=index)
```

Here, *data* can be many different things:

- a Python dict
- an ndarray
- a scalar value (like 5)

The passed **index** is a list of axis labels. Thus, this separates into a few cases depending on what **data** is:

#### From ndarray

If *data* is an ndarray, **index** must be the same length as **data**. If no index is passed, one will be created having values `[0, ..., len(data) - 1]`.

```
In [3]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [4]: s
Out[4]:
a      0.4691
b     -0.2829
```

(continues on next page)

(continued from previous page)

```
c    -1.5091
d    -1.1356
e     1.2121
dtype: float64

In [5]: s.index
Out[5]:
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [6]: pd.Series(np.random.randn(5))
Out[6]:
0    -0.1732
1     0.1192
2    -1.0442
3    -0.8618
4    -2.1046
dtype: float64
```

---

**Note:** pandas supports non-unique index values. If an operation that does not support duplicate index values is attempted, an exception will be raised at that time. The reason for being lazy is nearly all performance-based (there are many instances in computations, like parts of GroupBy, where the index is not used).

---

### From dict

Series can be instantiated from dicts:

```
In [7]: d = {'b' : 1, 'a' : 0, 'c' : 2}

In [8]: pd.Series(d)
Out[8]:
b    1
a    0
c    2
dtype: int64
```

---

**Note:** When the data is a dict, and an index is not passed, the `Series` index will be ordered by the dict's insertion order, if you're using Python version `>= 3.6` and Pandas version `>= 0.23`.

If you're using Python `< 3.6` or Pandas `< 0.23`, and an index is not passed, the `Series` index will be the lexically ordered list of dict keys.

---

In the example above, if you were on a Python version lower than 3.6 or a Pandas version lower than 0.23, the `Series` would be ordered by the lexical order of the dict keys (i.e. `['a', 'b', 'c']` rather than `['b', 'a', 'c']`).

If an index is passed, the values in data corresponding to the labels in the index will be pulled out.

```
In [9]: d = {'a' : 0., 'b' : 1., 'c' : 2.}

In [10]: pd.Series(d)
Out[10]:
a    0.0
b    1.0
c    2.0
```

(continues on next page)

(continued from previous page)

```
dtype: float64

In [11]: pd.Series(d, index=['b', 'c', 'd', 'a'])
Out[11]:
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

**Note:** NaN (not a number) is the standard missing data marker used in pandas.

### From scalar value

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**.

```
In [12]: pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[12]:
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

## 8.1.1 Series is ndarray-like

Series acts very similarly to a ndarray, and is a valid argument to most NumPy functions. However, operations such as slicing will also slice the index.

```
In [13]: s[0]
Out[13]: 0.46911229990718628

In [14]: s[:3]
Out[14]:
a    0.4691
b   -0.2829
c   -1.5091
dtype: float64

In [15]: s[s > s.median()]
Out[15]:
a    0.4691
e    1.2121
dtype: float64

In [16]: s[[4, 3, 1]]
Out[16]:
e    1.2121
d   -1.1356
b   -0.2829
```

(continues on next page)

(continued from previous page)

```
dtype: float64
```

```
In [17]: np.exp(s)
```

```

a      1.5986
b      0.7536
c      0.2211
d      0.3212
e      3.3606
dtype: float64

```

We will address array-based indexing in a separate *section*.

### 8.1.2 Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

```
In [18]: s['a']
```

```
Out[18]: 0.46911229990718628
```

```
In [19]: s['e'] = 12.
```

```
In [20]: s
```

Out [20] :

|   |        |
|---|--------|
| a | 0.4691 |
|---|--------|

|   |         |
|---|---------|
| b | -0.2829 |
|---|---------|

|   |         |
|---|---------|
| c | -1.5091 |
|---|---------|

|   |         |
|---|---------|
| d | -1.1356 |
|---|---------|

|   |         |
|---|---------|
| e | 12.0000 |
|---|---------|

```
dtype: float64
```

```
In [21]: 'e' in s
```

↪ True

```
In [22]: 'f' in s
```

```

// ...
return false;
}
}

```

If a label is not contained, an exception is raised:

```
>>> s['f']
```

```
KeyError: 'f'
```

Using the `get` method, a missing label will return `None` or specified default:

```
In [23]: s.get('f')
```

```
In [24]: s.get('f', np.nan)
```

```
Out[24]: nan
```

See also the *section on attribute access*.



### 8.1.3 Vectorized operations and label alignment with Series

When working with raw NumPy arrays, looping through value-by-value is usually not necessary. The same is true when working with Series in pandas. Series can also be passed into most NumPy methods expecting an ndarray.

```
In [25]: s + s
Out[25]:
a      0.9382
b     -0.5657
c     -3.0181
d     -2.2713
e      24.0000
dtype: float64

In [26]: s * 2
Out[26]:
a      0.9382
b     -0.5657
c     -3.0181
d     -2.2713
e      24.0000
dtype: float64

In [27]: np.exp(s)
Out[27]:
a      1.5986
b      0.7536
c      0.2211
d      0.3212
e    162754.7914
dtype: float64
```

A key difference between Series and ndarray is that operations between Series automatically align the data based on label. Thus, you can write computations without giving consideration to whether the Series involved have the same labels.

```
In [28]: s[1:] + s[:-1]
Out[28]:
a      NaN
b     -0.5657
c     -3.0181
d     -2.2713
e      NaN
dtype: float64
```

The result of an operation between unaligned Series will have the **union** of the indexes involved. If a label is not found in one Series or the other, the result will be marked as missing NaN. Being able to write code without doing any explicit data alignment grants immense freedom and flexibility in interactive data analysis and research. The integrated data alignment features of the pandas data structures set pandas apart from the majority of related tools for working with labeled data.

**Note:** In general, we chose to make the default result of operations between differently indexed objects yield the **union** of the indexes in order to avoid loss of information. Having an index label, though the data is missing, is typically important information as part of a computation. You of course have the option of dropping labels with

missing data via the **dropna** function.

### 8.1.4 Name attribute

Series can also have a name attribute:

```
In [29]: s = pd.Series(np.random.randn(5), name='something')
```

```
In [30]: s
```

Out[30]:

|   |         |
|---|---------|
| 0 | -0.4949 |
|---|---------|

|   |        |
|---|--------|
| 1 | 1.0718 |
|---|--------|

|   |        |
|---|--------|
| 2 | 0.7216 |
|---|--------|

|   |         |
|---|---------|
| 3 | -0.7068 |
|---|---------|

|   |         |
|---|---------|
| 4 | -1.0396 |
|---|---------|

```
Name: something, dtype: float64
```

```
In [31]: s.name
```

```
↳ 'something'
```

The Series `name` will be assigned automatically in many cases, in particular when taking 1D slices of `DataFrame` as you will see below.

New in version 0.18.0.

You can rename a Series with the `pandas.Series.rename()` method.

```
In [32]: s2 = s.rename("different")
```

```
In [33]: s2.name
```

```
Out[33]: 'different'
```

Note that `s` and `s2` refer to different objects.

## 8.2 DataFrame

**DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It is generally the most commonly used pandas object. Like Series, DataFrame accepts many different kinds of input:

- Dict of 1D ndarrays, lists, dicts, or Series
- 2-D numpy.ndarray
- **Structured or record** ndarray
- A Series
- **Another** DataFrame

Along with the data, you can optionally pass **index** (row labels) and **columns** (column labels) arguments. If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame. Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.

If axis labels are not passed, they will be constructed from the input data based on common sense rules.

**Note:** When the data is a dict, and `columns` is not specified, the `DataFrame` columns will be ordered by the dict's insertion order, if you are using Python version  $\geq 3.6$  and Pandas  $\geq 0.23$ .

If you are using Python  $< 3.6$  or Pandas  $< 0.23$ , and `columns` is not specified, the `DataFrame` columns will be the lexically ordered list of dict keys.

## 8.2.1 From dict of Series or dicts

The resulting **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will first be converted to Series. If no columns are passed, the columns will be the ordered list of dict keys.

```
In [34]: d = {'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
....:        'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])}
....:

In [35]: df = pd.DataFrame(d)

In [36]: df
Out[36]:
   one  two
a  1.0  1.0
b  2.0  2.0
c  3.0  3.0
d  NaN  4.0

In [37]: pd.DataFrame(d, index=['d', 'b', 'a'])
Out[37]:
   one  two
d  NaN  4.0
b  2.0  2.0
a  1.0  1.0

In [38]: pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
Out[38]:
   two three
d  4.0   NaN
b  2.0   NaN
a  1.0   NaN
```

The row and column labels can be accessed respectively by accessing the **index** and **columns** attributes:

**Note:** When a particular set of columns is passed along with a dict of data, the passed columns override the keys in the dict.

```
In [39]: df.index
Out[39]: Index(['a', 'b', 'c', 'd'], dtype='object')

In [40]: df.columns
Out[40]: Index(['one', 'two'], dtype='object')
```

## 8.2.2 From dict of ndarrays / lists

The ndarrays must all be the same length. If an index is passed, it must clearly also be the same length as the arrays. If no index is passed, the result will be `range(n)`, where `n` is the array length.

```
In [41]: d = {'one' : [1., 2., 3., 4.],
....:       'two' : [4., 3., 2., 1.]}
....:

In [42]: pd.DataFrame(d)
Out[42]:
   one two
0  1.0  4.0
1  2.0  3.0
2  3.0  2.0
3  4.0  1.0

In [43]: pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[43]:
   one two
a  1.0  4.0
b  2.0  3.0
c  3.0  2.0
d  4.0  1.0
```

## 8.2.3 From structured or record array

This case is handled identically to a dict of arrays.

```
In [44]: data = np.zeros((2,), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])

In [45]: data[:] = [(1, 2., 'Hello'), (2, 3., "World")]

In [46]: pd.DataFrame(data)
Out[46]:
   A    B    C
0  1  2.0 b'Hello'
1  2  3.0 b'World'

In [47]: pd.DataFrame(data, index=['first', 'second'])
Out[47]:
      A    B    C
first  1  2.0 b'Hello'
second 2  3.0 b'World'

In [48]: pd.DataFrame(data, columns=['C', 'A', 'B'])
Out[48]:
   C    A    B
0 b'Hello'  1  2.0
1 b'World'  2  3.0
```

**Note:** DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.

## 8.2.4 From a list of dicts

```
In [49]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [50]: pd.DataFrame(data2)
Out[50]:
   a  b    c
0  1  2  NaN
1  5 10 20.0

In [51]: pd.DataFrame(data2, index=['first', 'second'])
Out[51]:
      a  b    c
first  1  2  NaN
second 5 10 20.0

In [52]: pd.DataFrame(data2, columns=['a', 'b'])
Out[52]:
   a  b
0  1  2
1  5 10
```

## 8.2.5 From a dict of tuples

You can automatically create a multi-indexed frame by passing a tuples dictionary.

```
In [53]: pd.DataFrame({'a', 'b': {('A', 'B'): 1, ('A', 'C'): 2,
....:                             ('a', 'a'): 3, ('a', 'b'): 4,
....:                             ('a', 'c'): 5, ('a', 'd'): 6,
....:                             ('b', 'a'): 7, ('b', 'b'): 8,
....:                             ('b', 'c'): 9, ('b', 'd'): 10}})
Out[53]:
      a  b
A B  1.0  4.0  5.0  8.0 10.0
C  2.0  3.0  6.0  7.0  NaN
D  NaN  NaN  NaN  NaN  9.0
```

## 8.2.6 From a Series

The result will be a DataFrame with the same index as the input Series, and with one column whose name is the original name of the Series (only if no other column name provided).

### Missing Data

Much more will be said on this topic in the [Missing data](#) section. To construct a DataFrame with missing data, we use `np.nan` to represent missing values. Alternatively, you may pass a `numpy.MaskedArray` as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

## 8.2.7 Alternate Constructors

### DataFrame.from\_dict

`DataFrame.from_dict` takes a dict of dicts or a dict of array-like sequences and returns a `DataFrame`. It operates like the `DataFrame` constructor except for the `orient` parameter which is `'columns'` by default, but which can be set to `'index'` in order to use the dict keys as row labels.

```
In [54]: pd.DataFrame.from_dict(dict([('A', [1, 2, 3]), ('B', [4, 5, 6])]))
Out[54]:
```

|   | A | B |
|---|---|---|
| 0 | 1 | 4 |
| 1 | 2 | 5 |
| 2 | 3 | 6 |

If you pass `orient='index'`, the keys will be the row labels. In this case, you can also pass the desired column names:

```
In [55]: pd.DataFrame.from_dict(dict([( 'A', [1, 2, 3]), ( 'B', [4, 5, 6])]),
....:                             orient='index', columns=['one', 'two', 'three'])
....:
Out[55]:
```

|   | one | two | three |
|---|-----|-----|-------|
| A | 1   | 2   | 3     |
| B | 4   | 5   | 6     |

## DataFrame.from\_records

`DataFrame.from_records` takes a list of tuples or an ndarray with structured dtype. It works analogously to the normal `DataFrame` constructor, except that the resulting `DataFrame` index may be a specific field of the structured dtype. For example:

```
In [56]: data
Out[56]:
array([(1, 2., b'Hello'), (2, 3., b'World')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])

In [57]: pd.DataFrame.from_records(data, index='C')
//////////
↪
      A      B
C
b'Hello'  1  2.0
b'World'  2  3.0
```

### 8.2.8 Column selection, addition, deletion

You can treat a `DataFrame` semantically like a dict of like-indexed `Series` objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations:

```
In [58]: df['one']
Out[58]:
a      1.0
b      2.0
c      3.0
d      NaN
Name: one, dtype: float64

In [59]: df['three'] = df['one'] * df['two']
```

(continues on next page)

(continued from previous page)

```
In [60]: df['flag'] = df['one'] > 2
```

```
In [61]: df
```

```
Out[61]:
```

|   | one | two | three | flag  |
|---|-----|-----|-------|-------|
| a | 1.0 | 1.0 | 1.0   | False |
| b | 2.0 | 2.0 | 4.0   | False |
| c | 3.0 | 3.0 | 9.0   | True  |
| d | NaN | 4.0 | NaN   | False |

Columns can be deleted or popped like with a dict:

```
In [62]: del df['two']
```

```
In [63]: three = df.pop('three')
```

```
In [64]: df
```

```
Out[64]:
```

|   | one | flag  |
|---|-----|-------|
| a | 1.0 | False |
| b | 2.0 | False |
| c | 3.0 | True  |
| d | NaN | False |

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [65]: df['foo'] = 'bar'
```

```
In [66]: df
```

```
Out[66]:
```

|   | one | flag  | foo |
|---|-----|-------|-----|
| a | 1.0 | False | bar |
| b | 2.0 | False | bar |
| c | 3.0 | True  | bar |
| d | NaN | False | bar |

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [67]: df['one_trunc'] = df['one'][:2]
```

```
In [68]: df
```

```
Out[68]:
```

|   | one | flag  | foo | one_trunc |
|---|-----|-------|-----|-----------|
| a | 1.0 | False | bar | 1.0       |
| b | 2.0 | False | bar | 2.0       |
| c | 3.0 | True  | bar | NaN       |
| d | NaN | False | bar | NaN       |

You can insert raw ndarrays but their length must match the length of the DataFrame's index.

By default, columns get inserted at the end. The `insert` function is available to insert at a particular location in the columns:

```
In [69]: df.insert(1, 'bar', df['one'])
```

```
In [70]: df
```

(continues on next page)

(continued from previous page)

```
Out [70]:
```

|   | one | bar | flag  | foo | one_trunc |
|---|-----|-----|-------|-----|-----------|
| a | 1.0 | 1.0 | False | bar | 1.0       |
| b | 2.0 | 2.0 | False | bar | 2.0       |
| c | 3.0 | 3.0 | True  | bar | NaN       |
| d | NaN | NaN | False | bar | NaN       |

## 8.2.9 Assigning New Columns in Method Chains

Inspired by `dplyr`'s `mutate` verb, `DataFrame` has an `assign()` method that allows you to easily create new columns that are potentially derived from existing columns.

```
In [71]: iris = pd.read_csv('data/iris.data')
```

```
In [72]: iris.head()
```

```
Out [72]:
```

|   | SepalLength | SepalWidth | PetalLength | PetalWidth | Name        |
|---|-------------|------------|-------------|------------|-------------|
| 0 | 5.1         | 3.5        | 1.4         | 0.2        | Iris-setosa |
| 1 | 4.9         | 3.0        | 1.4         | 0.2        | Iris-setosa |
| 2 | 4.7         | 3.2        | 1.3         | 0.2        | Iris-setosa |
| 3 | 4.6         | 3.1        | 1.5         | 0.2        | Iris-setosa |
| 4 | 5.0         | 3.6        | 1.4         | 0.2        | Iris-setosa |

```
In [73]: (iris.assign(sepal_ratio = iris['SepalWidth'] / iris['SepalLength']))
```

```
.....: .head())
```

```
.....:
```

```
=====
```

```
↪
```

|   | SepalLength | SepalWidth | PetalLength | PetalWidth | Name        | sepal_ratio |
|---|-------------|------------|-------------|------------|-------------|-------------|
| 0 | 5.1         | 3.5        | 1.4         | 0.2        | Iris-setosa | 0.6863      |
| 1 | 4.9         | 3.0        | 1.4         | 0.2        | Iris-setosa | 0.6122      |
| 2 | 4.7         | 3.2        | 1.3         | 0.2        | Iris-setosa | 0.6809      |
| 3 | 4.6         | 3.1        | 1.5         | 0.2        | Iris-setosa | 0.6739      |
| 4 | 5.0         | 3.6        | 1.4         | 0.2        | Iris-setosa | 0.7200      |

In the example above, we inserted a precomputed value. We can also pass in a function of one argument to be evaluated on the `DataFrame` being assigned to.

```
In [74]: iris.assign(sepal_ratio = lambda x: (x['SepalWidth'] /
```

```
.....: x['SepalLength'])).head()
```

```
.....:
```

```
Out [74]:
```

|   | SepalLength | SepalWidth | PetalLength | PetalWidth | Name        | sepal_ratio |
|---|-------------|------------|-------------|------------|-------------|-------------|
| 0 | 5.1         | 3.5        | 1.4         | 0.2        | Iris-setosa | 0.6863      |
| 1 | 4.9         | 3.0        | 1.4         | 0.2        | Iris-setosa | 0.6122      |
| 2 | 4.7         | 3.2        | 1.3         | 0.2        | Iris-setosa | 0.6809      |
| 3 | 4.6         | 3.1        | 1.5         | 0.2        | Iris-setosa | 0.6739      |
| 4 | 5.0         | 3.6        | 1.4         | 0.2        | Iris-setosa | 0.7200      |

`assign` **always** returns a copy of the data, leaving the original `DataFrame` untouched.

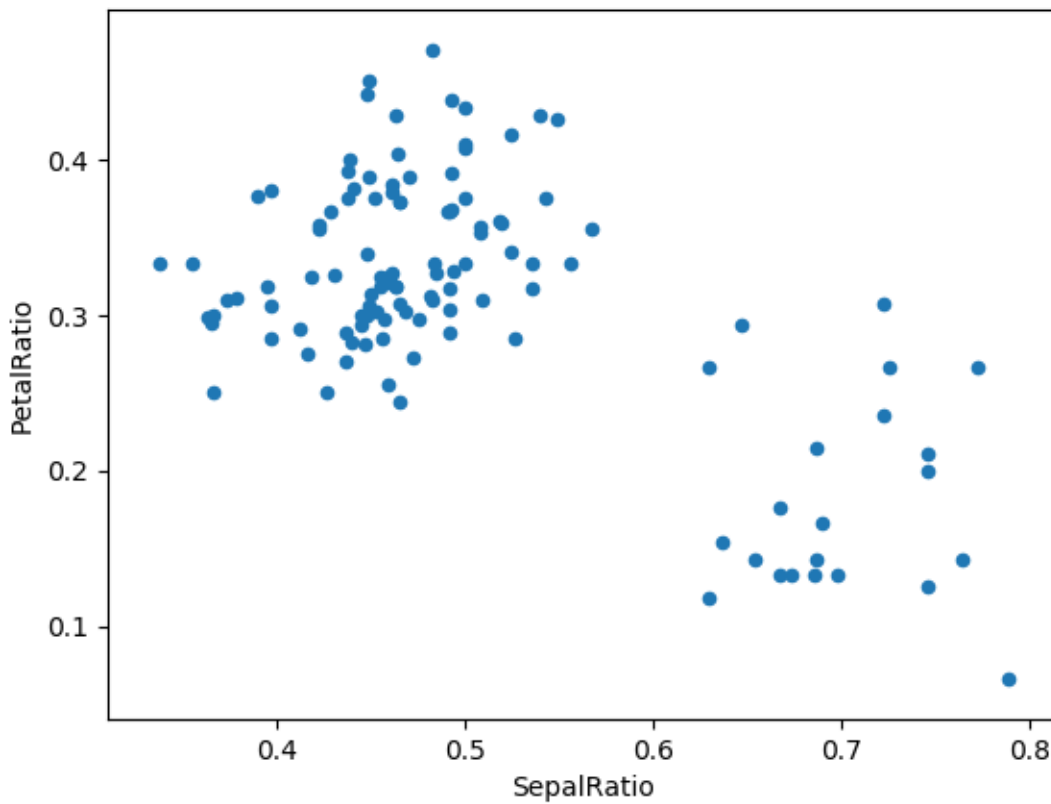
Passing a callable, as opposed to an actual value to be inserted, is useful when you don't have a reference to the `DataFrame` at hand. This is common when using `assign` in a chain of operations. For example, we can limit the `DataFrame` to just those observations with a Sepal Length greater than 5, calculate the ratio, and plot:



```

In [75]: (iris.query('SepalLength > 5')
.....:      .assign(SepalRatio = lambda x: x.SepalWidth / x.SepalLength,
.....:              PetalRatio = lambda x: x.PetalWidth / x.PetalLength)
.....:      .plot(kind='scatter', x='SepalRatio', y='PetalRatio'))
.....:
Out [75]: <matplotlib.axes._subplots.AxesSubplot at 0x1c259fd7b8>

```



Since a function is passed in, the function is computed on the DataFrame being assigned to. Importantly, this is the DataFrame that's been filtered to those rows with sepal length greater than 5. The filtering happens first, and then the ratio calculations. This is an example where we didn't have a reference to the *filtered* DataFrame available.

The function signature for `assign` is simply `**kwargs`. The keys are the column names for the new fields, and the values are either a value to be inserted (for example, a `Series` or NumPy array), or a function of one argument to be called on the DataFrame. A *copy* of the original DataFrame is returned, with the new values inserted.

Changed in version 0.23.0.

Starting with Python 3.6 the order of `**kwargs` is preserved. This allows for *dependent* assignment, where an expression later in `**kwargs` can refer to a column created earlier in the same `assign()`.

```

In [76]: dfa = pd.DataFrame({"A": [1, 2, 3],
.....:                      "B": [4, 5, 6]})
.....:

In [77]: dfa.assign(C=lambda x: x['A'] + x['B'],
.....:              D=lambda x: x['A'] + x['C'])

```

(continues on next page)

(continued from previous page)

```
.....:
Out [77]:
   A  B  C  D
0  1  4  5  6
1  2  5  7  9
2  3  6  9 12
```

In the second expression, `x[ 'C' ]` will refer to the newly created column, that's equal to `dfa[ 'A' ] + dfa[ 'B' ]`. To write code compatible with all versions of Python, split the assignment in two.

```
In [78]: dependent = pd.DataFrame({"A": [1, 1, 1]})

In [79]: (dependent.assign(A=lambda x: x[ 'A' ] + 1)
.....:               .assign(B=lambda x: x[ 'A' ] + 2))
.....:
Out [79]:
   A  B
0  2  4
1  2  4
2  2  4
```

**Warning:** Dependent assignment maybe subtly change the behavior of your code between Python 3.6 and older versions of Python.

If you wish write code that supports versions of python before and after 3.6, you'll need to take care when passing assign expressions that

- Updating an existing column
- Referring to the newly updated column in the same assign

For example, we'll update column "A" and then refer to it when creating "B".

```
>>> dependent = pd.DataFrame({"A": [1, 1, 1]})
>>> dependent.assign(A=lambda x: x["A"] + 1,
.....:               B=lambda x: x["A"] + 2)
```

For Python 3.5 and earlier the expression creating B refers to the "old" value of A, `[1, 1, 1]`. The output is then

```
   A  B
0  2  3
1  2  3
2  2  3
```

For Python 3.6 and later, the expression creating A refers to the "new" value of A, `[2, 2, 2]`, which results in

```
   A  B
0  2  4
1  2  4
2  2  4
```

## 8.2.10 Indexing / Selection

The basics of indexing are as follows:

```
In [80]: df.loc['b']
Out[80]:
one          2
bar          2
flag        False
foo          bar
one_trunc    2
Name: b, dtype: object
```

```
In [81]: df.iloc[2]
\\/////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↪
one          3
bar          3
flag         True
foo          bar
one_trunc    NaN
Name: c, dtype: object
```

```
In [82]: df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
```

```
In [83]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
```

```
In [84]: df + df2
```

Out [84]:

|   | A       | B       | C       | D   |
|---|---------|---------|---------|-----|
| 0 | 0.0457  | -0.0141 | 1.3809  | NaN |
| 1 | -0.9554 | -1.5010 | 0.0372  | NaN |
| 2 | -0.6627 | 1.5348  | -0.8597 | NaN |
| 3 | -2.4529 | 1.2373  | -0.1337 | NaN |
| 4 | 1.4145  | 1.9517  | -2.3204 | NaN |
| 5 | -0.4949 | -1.6497 | -1.0846 | NaN |
| 6 | -1.0476 | -0.7486 | -0.8055 | NaN |
| 7 | NaN     | NaN     | NaN     | NaN |
| 8 | NaN     | NaN     | NaN     | NaN |
| 9 | NaN     | NaN     | NaN     | NaN |

When doing an operation between DataFrame and Series, the default behavior is to align the Series **index** on the DataFrame **columns**, thus **broadcasting** row-wise. For example:

```
In [85]: df - df.iloc[0]
Out[85]:
```

|   | A       | B       | C       | D       |
|---|---------|---------|---------|---------|
| 0 | 0.0000  | 0.0000  | 0.0000  | 0.0000  |
| 1 | -1.3593 | -0.2487 | -0.4534 | -1.7547 |
| 2 | 0.2531  | 0.8297  | 0.0100  | -1.9912 |
| 3 | -1.3111 | 0.0543  | -1.7249 | -1.6205 |
| 4 | 0.5730  | 1.5007  | -0.6761 | 1.3673  |
| 5 | -1.7412 | 0.7820  | -1.2416 | -2.0531 |
| 6 | -1.2408 | -0.8696 | -0.1533 | 0.0004  |
| 7 | -0.7439 | 0.4110  | -0.9296 | -0.2824 |
| 8 | -1.1949 | 1.3207  | 0.2382  | -1.4826 |
| 9 | 2.2938  | 1.8562  | 0.7733  | -1.4465 |

In the special case of working with time series data, and the DataFrame index also contains dates, the broadcasting will be column-wise:

```
In [86]: index = pd.date_range('1/1/2000', periods=8)
In [87]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=list('ABC'))
```

```
In [88]: df
Out[88]:
```

|            | A       | B       | C       |
|------------|---------|---------|---------|
| 2000-01-01 | -1.2268 | 0.7698  | -1.2812 |
| 2000-01-02 | -0.7277 | -0.1213 | -0.0979 |
| 2000-01-03 | 0.6958  | 0.3417  | 0.9597  |
| 2000-01-04 | -1.1103 | -0.6200 | 0.1497  |
| 2000-01-05 | -0.7323 | 0.6877  | 0.1764  |
| 2000-01-06 | 0.4033  | -0.1550 | 0.3016  |
| 2000-01-07 | -2.1799 | -1.3698 | -0.9542 |
| 2000-01-08 | 1.4627  | -1.7432 | -0.8266 |

[illegible]

```
In [90]: df - df['A']
\
↪
2000-01-01 00:00:00 2000-01-02 00:00:00 2000-01-03 00:00:00 \
2000-01-01      NaN      NaN      NaN
2000-01-02      NaN      NaN      NaN
2000-01-03      NaN      NaN      NaN
2000-01-04      NaN      NaN      NaN
2000-01-05      NaN      NaN      NaN
2000-01-06      NaN      NaN      NaN
2000-01-07      NaN      NaN      NaN
2000-01-08      NaN      NaN      NaN

2000-01-04 00:00:00 ... 2000-01-08 00:00:00  A  B  C
2000-01-01      NaN ...      NaN NaN NaN NaN
2000-01-02      NaN ...      NaN NaN NaN NaN
2000-01-03      NaN ...      NaN NaN NaN NaN
2000-01-04      NaN ...      NaN NaN NaN NaN
```

(continues on next page)

```
2000-01-05      NaN ...      NaN NaN NaN NaN
2000-01-06      NaN ...      NaN NaN NaN NaN
2000-01-07      NaN ...      NaN NaN NaN NaN
2000-01-08      NaN ...      NaN NaN NaN NaN

[8 rows x 11 columns]
```

```
df = df['A']
```

```
df.sub(df['A'], axis=0)
```

Operations with scalars are just as you would expect:

Out [91] :

```
In [92]: 1 / df
```

```
In [93]: df ** 4
```

|            | A       | B      | C          |
|------------|---------|--------|------------|
| 2000-01-01 | 2.2653  | 0.3512 | 2.6948e+00 |
| 2000-01-02 | 0.2804  | 0.0002 | 9.1796e-05 |
| 2000-01-03 | 0.2344  | 0.0136 | 8.4838e-01 |
| 2000-01-04 | 1.5199  | 0.1477 | 5.0286e-04 |
| 2000-01-05 | 0.2876  | 0.2237 | 9.6924e-04 |
| 2000-01-06 | 0.0265  | 0.0006 | 8.2769e-03 |
| 2000-01-07 | 22.5795 | 3.5212 | 8.2903e-01 |
| 2000-01-08 | 4.5774  | 9.2332 | 4.6683e-01 |

Boolean operators work as well:

```
In [94]: df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)
In [95]: df2 = pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
In [96]: df1 & df2
Out[96]:
   a      b
0  False False
1  False  True
2   True False

In [97]: df1 | df2
Out[97]:
   a      b
0   True  True
1   True  True
2   True  True

In [98]: df1 ^ df2
Out[98]:
   a      b
0   True  True
1   True False
2  False  True

In [99]: ~df1
Out[99]:
   a      b
0 False  True
1  True False
2 False False
```

## 8.2.12 Transposing

To transpose, access the `T` attribute (also the `transpose` function), similar to an `ndarray`:

```
# only show the first 5 rows
In [100]: df[:5].T
Out[100]:
2000-01-01  2000-01-02  2000-01-03  2000-01-04  2000-01-05
A      -1.2268    -0.7277     0.6958    -1.1103    -0.7323
B       0.7698    -0.1213     0.3417    -0.6200     0.6877
C      -1.2812    -0.0979     0.9597     0.1497     0.1764
```

## 8.2.13 DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (`log`, `exp`, `sqrt`, ...) and various other NumPy functions can be used with no issues on `DataFrame`, assuming the data within are numeric:

```
In [101]: np.exp(df)
Out[101]:
```

(continues on next page)

(continued from previous page)

```

      A      B      C
2000-01-01  0.2932  2.1593  0.2777
2000-01-02  0.4830  0.8858  0.9068
2000-01-03  2.0053  1.4074  2.6110
2000-01-04  0.3294  0.5380  1.1615
2000-01-05  0.4808  1.9892  1.1930
2000-01-06  1.4968  0.8565  1.3521
2000-01-07  0.1131  0.2541  0.3851
2000-01-08  4.3176  0.1750  0.4375

```

```
In [102]: np.asarray(df)
```

```

////////////////////////////////////
↪
array([[ -1.2268,   0.7698,  -1.2812],
       [ -0.7277,  -0.1213,  -0.0979],
       [  0.6958,   0.3417,   0.9597],
       [ -1.1103,  -0.62   ,   0.1497],
       [ -0.7323,   0.6877,   0.1764],
       [  0.4033,  -0.155  ,   0.3016],
       [ -2.1799,  -1.3698,  -0.9542],
       [  1.4627,  -1.7432,  -0.8266]])

```

The dot method on DataFrame implements matrix multiplication:

```
In [103]: df.T.dot(df)
```

```
Out[103]:
```

```

      A      B      C
A  11.3419 -0.0598  3.0080
B  -0.0598  6.5206  2.0833
C   3.0080  2.0833  4.3105

```

Similarly, the dot method on Series implements dot product:

```
In [104]: s1 = pd.Series(np.arange(5,10))
```

```
In [105]: s1.dot(s1)
```

```
Out[105]: 255
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics are quite different in places from a matrix.

## 8.2.14 Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using `info()`. (Here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [106]: baseball = pd.read_csv('data/baseball.csv')
```

```
In [107]: print(baseball)
```

```

   id  player  year  stint  ...  hbp  sh  sf  gidp
0  88641  womacto01  2006     2  ...  0.0  3.0  0.0   0.0
1  88643  schilcu01  2006     1  ...  0.0  0.0  0.0   0.0
..  ...      ...      ...  ...  ...  ...  ...  ...
98  89533  aloumo01  2007     1  ...  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007     1  ...  0.0  0.0  0.0   0.0

```

(continues on next page)

(continued from previous page)

```
[100 rows x 23 columns]

In [108]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
id            100 non-null int64
player       100 non-null object
year         100 non-null int64
stint        100 non-null int64
team         100 non-null object
lg           100 non-null object
g            100 non-null int64
ab           100 non-null int64
r            100 non-null int64
h            100 non-null int64
X2b          100 non-null int64
X3b          100 non-null int64
hr           100 non-null int64
rbi          100 non-null float64
sb           100 non-null float64
cs           100 non-null float64
bb           100 non-null int64
so           100 non-null float64
ibb          100 non-null float64
hbp          100 non-null float64
sh           100 non-null float64
sf           100 non-null float64
gidp         100 non-null float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.0+ KB
```

However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won't always fit the console width:

| In [109]: print(baseball.iloc[-20:, :12].to_string()) |       |           |      |       |      |    |     |     |    |     |     |     |
|---|-------|-----------|------|-------|------|----|-----|-----|----|-----|-----|-----|
|   | id    | player    | year | stint | team | lg | g   | ab  | r  | h   | X2b | X3b |
| 80  | 89474 | finlest01 | 2007 | 1     | COL  | NL | 43  | 94  | 9  | 17  | 3   | 0   |
| 81  | 89480 | embreal01 | 2007 | 1     | OAK  | AL | 4   | 0   | 0  | 0   | 0   | 0   |
| 82  | 89481 | edmonji01 | 2007 | 1     | SLN  | NL | 117 | 365 | 39 | 92  | 15  | 2   |
| 83  | 89482 | easleda01 | 2007 | 1     | NYN  | NL | 76  | 193 | 24 | 54  | 6   | 0   |
| 84  | 89489 | delgaca01 | 2007 | 1     | NYN  | NL | 139 | 538 | 71 | 139 | 30  | 0   |
| 85  | 89493 | cormirh01 | 2007 | 1     | CIN  | NL | 6   | 0   | 0  | 0   | 0   | 0   |
| 86  | 89494 | coninje01 | 2007 | 2     | NYN  | NL | 21  | 41  | 2  | 8   | 2   | 0   |
| 87  | 89495 | coninje01 | 2007 | 1     | CIN  | NL | 80  | 215 | 23 | 57  | 11  | 1   |
| 88  | 89497 | clemero02 | 2007 | 1     | NYA  | AL | 2   | 2   | 0  | 1   | 0   | 0   |
| 89  | 89498 | claytro01 | 2007 | 2     | BOS  | AL | 8   | 6   | 1  | 0   | 0   | 0   |
| 90  | 89499 | claytro01 | 2007 | 1     | TOR  | AL | 69  | 189 | 23 | 48  | 14  | 0   |
| 91  | 89501 | cirilje01 | 2007 | 2     | ARI  | NL | 28  | 40  | 6  | 8   | 4   | 0   |
| 92  | 89502 | cirilje01 | 2007 | 1     | MIN  | AL | 50  | 153 | 18 | 40  | 9   | 2   |
| 93  | 89521 | bondsba01 | 2007 | 1     | SFN  | NL | 126 | 340 | 75 | 94  | 14  | 0   |
| 94  | 89523 | biggicr01 | 2007 | 1     | HOU  | NL | 141 | 517 | 68 | 130 | 31  | 3   |
| 95  | 89525 | benitar01 | 2007 | 2     | FLO  | NL | 34  | 0   | 0  | 0   | 0   | 0   |
| 96  | 89526 | benitar01 | 2007 | 1     | SFN  | NL | 19  | 0   | 0  | 0   | 0   | 0   |

(continues on next page)



(continued from previous page)

|    |       |           |      |   |     |    |     |     |    |     |    |   |
|----|-------|-----------|------|---|-----|----|-----|-----|----|-----|----|---|
| 97 | 89530 | ausmubr01 | 2007 | 1 | HOU | NL | 117 | 349 | 38 | 82  | 16 | 3 |
| 98 | 89533 | aloumo01  | 2007 | 1 | NYN | NL | 87  | 328 | 51 | 112 | 19 | 1 |
| 99 | 89534 | alomasa02 | 2007 | 1 | NYN | NL | 8   | 22  | 1  | 3   | 1  | 0 |

Wide DataFrames will be printed across multiple rows by default:

```
In [110]: pd.DataFrame(np.random.randn(3, 12))
Out[110]:
```

|   | 0         | 1         | 2         | 3        | 4        | ... | 7         | 8         | ... | 11  |
|---|-----------|-----------|-----------|----------|----------|-----|-----------|-----------|-----|-----|
| 0 | -0.345352 | 1.314232  | 0.690579  | 0.995761 | 2.396780 | ... | -0.317441 | -1.236269 | 0.  | ... |
| 1 | -2.182937 | 0.380396  | 0.084844  | 0.432390 | 1.519970 | ... | 0.274230  | 0.132885  | -0. | ... |
| 2 | 0.206053  | -0.251905 | -2.213588 | 1.063327 | 1.266143 | ... | 0.408204  | -1.048089 | -0. | ... |

[3 rows x 12 columns]

You can change how much to print on a single row by setting the `display.width` option:

```
In [111]: pd.set_option('display.width', 40) # default is 80
In [112]: pd.DataFrame(np.random.randn(3, 12))
Out[112]:
```

|   | 0        | 1         | 2         | 3         | 4         | ... | 7         | 8         | ... | 11  |
|---|----------|-----------|-----------|-----------|-----------|-----|-----------|-----------|-----|-----|
| 0 | 1.262731 | 1.289997  | 0.082423  | -0.055758 | 0.536580  | ... | -0.034571 | -2.484478 | -0. | ... |
| 1 | 1.126203 | -0.977349 | 1.474071  | -0.064034 | -1.282782 | ... | 0.441153  | 2.353925  | 0.  | ... |
| 2 | 0.758527 | 1.729689  | -0.964980 | -0.845696 | -1.340896 | ... | 1.682706  | -1.717693 | 0.  | ... |

[3 rows x 12 columns]

You can adjust the max width of the individual columns by setting `display.max_colwidth`

```
In [113]: datafile={'filename': ['filename_01', 'filename_02'],
.....:               'path': ["media/user_name/storage/folder_01/filename_01",
.....:                        "media/user_name/storage/folder_02/filename_02"]}
In [114]: pd.set_option('display.max_colwidth', 30)
In [115]: pd.DataFrame(datafile)
Out[115]:
```

|   | filename    | path                          |
|---|-------------|-------------------------------|
| 0 | filename_01 | media/user_name/storage/fo... |
| 1 | filename_02 | media/user_name/storage/fo... |

```
In [116]: pd.set_option('display.max_colwidth', 100)
In [117]: pd.DataFrame(datafile)
Out[117]:
```

|   | filename    | path  |
|---|-------------|---|
| 0 | filename_01 | media/user_name/storage/folder_01/filename_01 |
| 1 | filename_02 | media/user_name/storage/folder_02/filename_02 |

(continues on next page)

(continued from previous page)

```
0 filename_01 media/user_name/storage/folder_01/filename_01
1 filename_02 media/user_name/storage/folder_02/filename_02
```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

### 8.2.15 DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [118]: df = pd.DataFrame({'foo1' : np.random.randn(5),
.....:                       'foo2' : np.random.randn(5)})
.....:
```

```
In [119]: df
```

Out [119] :

|   | foo1      | foo2      |
|---|-----------|-----------|
| 0 | 1.171216  | -0.858447 |
| 1 | 0.520260  | 0.306996  |
| 2 | -1.197071 | -0.028665 |
| 3 | -1.066969 | 0.384316  |
| 4 | -0.303421 | 1.574159  |

```
In [120]: df.foo1
```

```
0    1.171216
1    0.520260
2   -1.197071
3   -1.066969
4   -0.303421
Name: fool, dtype: float64
```

The columns are also connected to the `IPython` completion mechanism so they can be tab-completed:

```
In [5]: df.fo<TAB>
df.foo1  df.foo2
```

## 8.3 Panel

**Warning:** In 0.20.0, `Panel` is deprecated and will be removed in a future version. See the section [Deprecate Panel](#).

Panel is a somewhat less-used, but still important container for 3-dimensional data. The term **panel data** is derived from econometrics and is partially responsible for the name pandas: pan(el)-da(ta)-s. The names for the 3 axes are intended to give some semantic meaning to describing operations involving panel data and, in particular, econometric analysis of panel data. However, for the strict purposes of slicing and dicing a collection of DataFrame objects, you may find the axis names slightly arbitrary:

- **items:** axis 0, each item corresponds to a DataFrame contained inside
- **major axis:** axis 1, it is the **index** (rows) of each of the DataFrames

- **minor\_axis**: axis 2, it is the **columns** of each of the DataFrames

Construction of Panels works about like you would expect:

### 8.3.1 From 3D ndarray with optional axis labels

```
In [121]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
.....:                  major_axis=pd.date_range('1/1/2000', periods=5),
.....:                  minor_axis=['A', 'B', 'C', 'D'])
.....:

In [122]: wp
Out[122]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

### 8.3.2 From dict of DataFrame objects

```
In [123]: data = {'Item1' : pd.DataFrame(np.random.randn(4, 3)),
.....:            'Item2' : pd.DataFrame(np.random.randn(4, 2))}
.....:

In [124]: pd.Panel(data)
Out[124]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 0 to 3
Minor_axis axis: 0 to 2
```

Note that the values in the dict need only be **convertible to DataFrame**. Thus, they can be any of the other valid inputs to DataFrame as per above.

One helpful factory method is `Panel.from_dict`, which takes a dictionary of DataFrames as above, and the following named parameters:

| Parameter              | Default            | Description  |
|------------------------|--------------------|--|
| <code>intersect</code> | <code>False</code> | drops elements whose indices do not align                        |
| <code>orient</code>    | <code>items</code> | use <code>minor</code> to use DataFrames' columns as panel items |

For example, compare to the construction above:

```
In [125]: pd.Panel.from_dict(data, orient='minor')
Out[125]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 2 (minor_axis)
Items axis: 0 to 2
Major_axis axis: 0 to 3
Minor_axis axis: Item1 to Item2
```

`Orient` is especially useful for mixed-type DataFrames. If you pass a dict of DataFrame objects with mixed-type columns, all of the data will get upcasted to `dtype=object` unless you pass `orient='minor'`:

```

In [126]: df = pd.DataFrame({'a': ['foo', 'bar', 'baz'],
.....:                      'b': np.random.randn(3)})
.....:

In [127]: df
Out[127]:
   a      b
0  foo -0.308853
1  bar -0.681087
2  baz  0.377953

In [128]: data = {'item1': df, 'item2': df}

In [129]: panel = pd.Panel.from_dict(data, orient='minor')

In [130]: panel['a']
Out[130]:
   item1 item2
0   foo   foo
1   bar   bar
2   baz   baz

In [131]: panel['b']
Out[131]:
   item1 item2
0 -0.308853 -0.308853
1 -0.681087 -0.681087
2  0.377953  0.377953

In [132]: panel['b'].dtypes
Out[132]:
item1    float64
item2    float64
dtype: object

```

**Note:** Panel, being less commonly used than Series and DataFrame, has been slightly neglected feature-wise. A number of methods and options available in DataFrame are not available in Panel.

### 8.3.3 From DataFrame using `to_panel` method

`to_panel` converts a DataFrame with a two-level index to a Panel.

```

In [133]: midx = pd.MultiIndex(levels=[['one', 'two'], ['x', 'y']], labels=[[1, 1, 0, 0],
.....:                               [1, 0, 1, 0]])
.....:

In [134]: df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)

In [135]: df.to_panel()
Out[135]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: A to B

```

(continues on next page)

(continued from previous page)

```
Major_axis axis: one to two
Minor_axis axis: x to y
```

### 8.3.4 Item selection / addition / deletion

Similar to DataFrame functioning as a dict of Series, Panel is like a dict of DataFrames:

```
In [136]: wp['Item1']
Out[136]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | 1.588931  | 0.476720  | 0.473424  | -0.242861 |
| 2000-01-02 | -0.014805 | -0.284319 | 0.650776  | -1.461665 |
| 2000-01-03 | -1.137707 | -0.891060 | -0.693921 | 1.613616  |
| 2000-01-04 | 0.464000  | 0.227371  | -0.496922 | 0.306389  |
| 2000-01-05 | -2.290613 | -1.134623 | -1.561819 | -0.260838 |

```
In [137]: wp['Item3'] = wp['Item1'] / wp['Item2']
```

The API for insertion and deletion is the same as for DataFrame. And as with DataFrame, if the item is a valid Python identifier, you can access it as an attribute and tab-complete it in IPython.

### 8.3.5 Transposing

A Panel can be rearranged using its `transpose` method (which does not make a copy by default unless the data are heterogeneous):

```
In [138]: wp.transpose(2, 0, 1)
Out[138]:
```

```
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 5 (minor_axis)
Items axis: A to D
Major_axis axis: Item1 to Item3
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```

### 8.3.6 Indexing / Selection

| Operation                     | Syntax                        | Result    |
|-------------------------------|-------------------------------|-----------|
| Select item                   | <code>wp[item]</code>         | DataFrame |
| Get slice at major_axis label | <code>wp.major_xs(val)</code> | DataFrame |
| Get slice at minor_axis label | <code>wp.minor_xs(val)</code> | DataFrame |

For example, using the earlier example data, we could do:

```
In [139]: wp['Item1']
Out[139]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | 1.588931  | 0.476720  | 0.473424  | -0.242861 |
| 2000-01-02 | -0.014805 | -0.284319 | 0.650776  | -1.461665 |
| 2000-01-03 | -1.137707 | -0.891060 | -0.693921 | 1.613616  |
| 2000-01-04 | 0.464000  | 0.227371  | -0.496922 | 0.306389  |

(continues on next page)

(continued from previous page)

```
2000-01-05 -2.290613 -1.134623 -1.561819 -0.260838
```

```
In [140]: wp.major_xs(wp.major_axis[2])
```

```

////////////////////////////////////
↪
      Item1      Item2      Item3
A -1.137707  0.800193 -1.421791
B -0.891060  0.782098 -1.139320
C -0.693921 -1.069094  0.649074
D  1.613616 -1.099248 -1.467927

```

```
In [141]: wp.minor_axis
```

```

////////////////////////////////////
↪Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
In [142]: wp.minor_xs('C')
```

```

////////////////////////////////////
↪
      Item1      Item2      Item3
2000-01-01  0.473424 -0.902937 -0.524316
2000-01-02  0.650776 -1.144073 -0.568824
2000-01-03 -0.693921 -1.069094  0.649074
2000-01-04 -0.496922  0.661084 -0.751678
2000-01-05 -1.561819 -1.056652  1.478083

```

### 8.3.7 Squeezing

Another way to change the dimensionality of an object is to squeeze a 1-len object, similar to `wp['Item1']`.

```
In [143]: wp.reindex(items=['Item1']).squeeze()
```

```
Out[143]:
      A      B      C      D
2000-01-01  1.588931  0.476720  0.473424 -0.242861
2000-01-02 -0.014805 -0.284319  0.650776 -1.461665
2000-01-03 -1.137707 -0.891060 -0.693921  1.613616
2000-01-04  0.464000  0.227371 -0.496922  0.306389
2000-01-05 -2.290613 -1.134623 -1.561819 -0.260838
```

```
In [144]: wp.reindex(items=['Item1'], minor=['B']).squeeze()
```

```

////////////////////////////////////
↪
2000-01-01    0.476720
2000-01-02   -0.284319
2000-01-03   -0.891060
2000-01-04    0.227371
2000-01-05   -1.134623
Freq: D, Name: B, dtype: float64

```

### 8.3.8 Conversion to DataFrame

A Panel can be represented in 2D form as a hierarchically indexed DataFrame. See the section [hierarchical indexing](#) for more on this. To convert a Panel to a DataFrame, use the `to_frame` method:

```
In [145]: panel = pd.Panel(np.random.randn(3, 5, 4), items=['one', 'two', 'three'],
.....:                    major_axis=pd.date_range('1/1/2000', periods=5),
.....:                    minor_axis=['a', 'b', 'c', 'd'])
.....:
```

```
In [146]: panel.to_frame()
```

```
Out[146]:
```

|            |            | one       | two       | three     |           |
|------------|------------|-----------|-----------|-----------|-----------|
| major      | minor      |           |           |           |           |
|            | 2000-01-01 | a         | 0.493672  | 1.219492  | -1.290493 |
|            |            | b         | -2.461467 | 0.062297  | 0.787872  |
|            |            | c         | -1.553902 | -0.110388 | 1.515707  |
| 2000-01-02 | d          | 2.015523  | -1.184357 | -0.276487 |           |
|            | a          | -1.833722 | -0.558081 | -0.223762 |           |
|            | b          | 1.771740  | 0.077849  | 1.397431  |           |
|            | c          | -0.670027 | 0.629498  | 1.503874  |           |
| 2000-01-03 | d          | 0.049307  | -1.035260 | -0.478905 |           |
|            | a          | -0.521493 | -0.438229 | -0.135950 |           |
|            | b          | -3.201750 | 0.503703  | -0.730327 |           |
|            | c          | 0.792716  | 0.413086  | -0.033277 |           |
| 2000-01-04 | d          | 0.146111  | -1.139050 | 0.281151  |           |
|            | a          | 1.903247  | 0.660342  | -1.298915 |           |
|            | b          | -0.747169 | 0.464794  | -2.819487 |           |
|            | c          | -0.309038 | -0.309337 | -0.851985 |           |
| 2000-01-05 | d          | 0.393876  | -0.649593 | -1.106952 |           |
|            | a          | 1.861468  | 0.683758  | -0.937731 |           |
|            | b          | 0.936527  | -0.643834 | -1.537770 |           |
|            | c          | 1.255746  | 0.421287  | 0.555759  |           |
|            | d          | -2.655452 | 1.032814  | -2.277282 |           |

## 8.4 Deprecate Panel

Over the last few years, pandas has increased in both breadth and depth, with new features, datatype support, and manipulation routines. As a result, supporting efficient indexing and functional routines for Series, DataFrame and Panel has contributed to an increasingly fragmented and difficult-to-understand codebase.

The 3-D structure of a Panel is much less common for many types of data analysis, than the 1-D of the Series or the 2-D of the DataFrame. Going forward it makes sense for pandas to focus on these areas exclusively.

Oftentimes, one can simply use a MultiIndex DataFrame for easily working with higher dimensional data.

In addition, the xarray package was built from the ground up, specifically in order to support the multi-dimensional analysis that is one of Panel's main usecases. [Here is a link to the xarray panel-transition documentation.](#)

```
In [147]: p = tm.makePanel()
```

```
In [148]: p
```

```
Out[148]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 30 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-02-11 00:00:00
Minor_axis axis: A to D
```

Convert to a MultiIndex DataFrame.

```
In [149]: p.to_frame()
```

```
Out [149]:
```

|            |       | ItemA     | ItemB     | ItemC     |
|------------|-------|-----------|-----------|-----------|
| major      | minor |           |           |           |
| 2000-01-03 | A     | -0.390201 | -1.624062 | -0.605044 |
|            | B     | 1.562443  | 0.483103  | 0.583129  |
|            | C     | -1.085663 | 0.768159  | -0.273458 |
|            | D     | 0.136235  | -0.021763 | -0.700648 |
| 2000-01-04 | A     | 1.207122  | -0.758514 | 0.878404  |
|            | B     | 0.763264  | 0.061495  | -0.876690 |
|            | C     | -1.114738 | 0.225441  | -0.335117 |
|            | D     | 0.886313  | -0.047152 | -1.166607 |
| 2000-01-05 | A     | 0.178690  | -0.560859 | -0.921485 |
|            | B     | 0.162027  | 0.240767  | -1.919354 |
|            | C     | -0.058216 | 0.543294  | -0.476268 |
|            | D     | -1.350722 | 0.088472  | -0.367236 |
| 2000-01-06 | A     | -1.004168 | -0.589005 | -0.200312 |
|            | B     | -0.902704 | 0.782413  | -0.572707 |
|            | C     | -0.486768 | 0.771931  | -1.765602 |
|            | D     | -0.886348 | -0.857435 | 1.296674  |
| 2000-01-07 | A     | -1.377627 | -1.070678 | 0.522423  |
|            | B     | 1.106010  | 0.628462  | -1.736484 |
|            | C     | 1.685148  | -0.968145 | 0.578223  |
|            | D     | -1.013316 | -2.503786 | 0.641385  |
| 2000-01-10 | A     | 0.499281  | -1.681101 | 0.722511  |
|            | B     | -0.199234 | -0.880627 | -1.335113 |
|            | C     | 0.112572  | -1.176383 | 0.242697  |
|            | D     | 1.920906  | -1.058041 | -0.779432 |
| 2000-01-11 | A     | -1.405256 | 0.403776  | -1.702486 |
|            | B     | 0.458265  | 0.777575  | -1.244471 |
|            | C     | -1.495309 | -3.192716 | 0.208129  |
|            | D     | -0.388231 | -0.657981 | 0.602456  |
| 2000-01-12 | A     | 0.162565  | 0.609862  | -0.709535 |
|            | B     | 0.491048  | -0.779367 | 0.347339  |
| ...        |       | ...       | ...       | ...       |
| 2000-02-02 | C     | -0.303961 | -0.463752 | -0.288962 |
|            | D     | 0.104050  | 1.116086  | 0.506445  |
| 2000-02-03 | A     | -2.338595 | -0.581967 | -0.801820 |
|            | B     | -0.557697 | -0.033731 | -0.176382 |
|            | C     | 0.625555  | -0.055289 | 0.875359  |
|            | D     | 0.174068  | -0.443915 | 1.626369  |
| 2000-02-04 | A     | -0.374279 | -1.233862 | -0.915751 |
|            | B     | 0.381353  | -1.108761 | -1.970108 |
|            | C     | -0.059268 | -0.360853 | -0.614618 |
|            | D     | -0.439461 | -0.200491 | 0.429518  |
| 2000-02-07 | A     | -2.359958 | -3.520876 | -0.288156 |
|            | B     | 1.337122  | -0.314399 | -1.044208 |
|            | C     | 0.249698  | 0.728197  | 0.565375  |
|            | D     | -0.741343 | 1.092633  | 0.013910  |
| 2000-02-08 | A     | -1.157886 | 0.516870  | -1.199945 |
|            | B     | -1.531095 | -0.860626 | -0.821179 |
|            | C     | 1.103949  | 1.326768  | 0.068184  |
|            | D     | -0.079673 | -1.675194 | -0.458272 |
| 2000-02-09 | A     | -0.551865 | 0.343125  | -0.072869 |
|            | B     | 1.331458  | 0.370397  | -1.914267 |
|            | C     | -1.087532 | 0.208927  | 0.788871  |
|            | D     | -0.922875 | 0.437234  | -1.531004 |

(continues on next page)



(continued from previous page)

```

2000-02-10 A      1.592673  2.137827 -1.828740
           B      -0.571329 -1.761442 -0.826439
           C      1.998044  0.292058 -0.280343
           D      0.303638  0.388254 -0.500569
2000-02-11 A      1.559318  0.452429 -1.716981
           B      -0.026671 -0.899454  0.124808
           C      -0.244548 -2.019610  0.931536
           D      -0.917368  0.479630  0.870690

```

```
[120 rows x 3 columns]
```

Alternatively, one can convert to an xarray DataArray.

```

In [150]: p.to_xarray()
Out [150]:
<xarray.DataArray (items: 3, major_axis: 30, minor_axis: 4)>
array([[[ -0.390201,  1.562443, -1.085663,  0.136235],
        [ 1.207122,  0.763264, -1.114738,  0.886313],
        ...,
        [ 1.592673, -0.571329,  1.998044,  0.303638],
        [ 1.559318, -0.026671, -0.244548, -0.917368]],

       [[ -1.624062,  0.483103,  0.768159, -0.021763],
        [-0.758514,  0.061495,  0.225441, -0.047152],
        ...,
        [ 2.137827, -1.761442,  0.292058,  0.388254],
        [ 0.452429, -0.899454, -2.01961 ,  0.47963 ]],

       [[ -0.605044,  0.583129, -0.273458, -0.700648],
        [ 0.878404, -0.87669 , -0.335117, -1.166607],
        ...,
        [-1.82874 , -0.826439, -0.280343, -0.500569],
        [-1.716981,  0.124808,  0.931536,  0.87069 ]]])
Coordinates:
  * items          (items) object 'ItemA' 'ItemB' 'ItemC'
  * major_axis     (major_axis) datetime64[ns] 2000-01-03 2000-01-04 2000-01-05 ...
  * minor_axis     (minor_axis) object 'A' 'B' 'C' 'D'

```

You can see the full-documentation for the [xarray](#) package.



## ESSENTIAL BASIC FUNCTIONALITY

Here we discuss a lot of the essential functionality common to the pandas data structures. Here's how to create some of the objects used in the examples from the previous section:

```
In [1]: index = pd.date_range('1/1/2000', periods=8)

In [2]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
...:                      columns=['A', 'B', 'C'])
...:
...:

In [4]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
...:                 major_axis=pd.date_range('1/1/2000', periods=5),
...:                 minor_axis=['A', 'B', 'C', 'D'])
...:
...:
```

### 9.1 Head and Tail

To view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```
In [5]: long_series = pd.Series(np.random.randn(1000))

In [6]: long_series.head()
Out[6]:
0    0.229453
1    0.304418
2    0.736135
3   -0.859631
4   -0.424100
dtype: float64

In [7]: long_series.tail(3)
Out[7]:
997   -0.351587
998    1.136249
999   -0.448789
dtype: float64
```

## 9.2 Attributes and the raw ndarray(s)

pandas objects have a number of attributes enabling you to access the metadata

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- Axis labels
  - **Series**: *index* (only axis)
  - **DataFrame**: *index* (rows) and *columns*
  - **Panel**: *items*, *major\_axis*, and *minor\_axis*

Note, these attributes can be safely assigned to!

```
In [8]: df[:2]
Out[8]:
```

|            | A         | B         | C        |
|------------|-----------|-----------|----------|
| 2000-01-01 | 0.048869  | -1.360687 | -0.47901 |
| 2000-01-02 | -0.859661 | -0.231595 | -0.52775 |

```
In [9]: df.columns = [x.lower() for x in df.columns]

In [10]: df
Out[10]:
```

|            | a         | b         | c         |
|------------|-----------|-----------|-----------|
| 2000-01-01 | 0.048869  | -1.360687 | -0.479010 |
| 2000-01-02 | -0.859661 | -0.231595 | -0.527750 |
| 2000-01-03 | -1.296337 | 0.150680  | 0.123836  |
| 2000-01-04 | 0.571764  | 1.555563  | -0.823761 |
| 2000-01-05 | 0.535420  | -1.032853 | 1.469725  |
| 2000-01-06 | 1.304124  | 1.449735  | 0.203109  |
| 2000-01-07 | -1.032011 | 0.969818  | -0.962723 |
| 2000-01-08 | 1.382083  | -0.938794 | 0.669142  |

To get the actual data inside a data structure, one need only access the **values** property:

```
In [11]: s.values
Out[11]: array([-1.9339,  0.3773,  0.7341,  2.1416, -0.0112])

In [12]: df.values
Out[12]:
```

```
array([[ 0.0489, -1.3607, -0.479 ],
       [-0.8597, -0.2316, -0.5278],
       [-1.2963,  0.1507,  0.1238],
       [ 0.5718,  1.5556, -0.8238],
       [ 0.5354, -1.0329,  1.4697],
       [ 1.3041,  1.4497,  0.2031],
       [-1.032 ,  0.9698, -0.9627],
       [ 1.3821, -0.9388,  0.6691]])

In [13]: wp.values
Out[13]:
```

```
array([[[-0.4336, -0.2736,  0.6804, -0.3084],
        [-0.2761, -1.8212, -1.9936, -1.9274],
        [-2.0279,  1.625 ,  0.5511,  3.0593],
        [ 0.4553, -0.0307,  0.9357,  1.0612],
```

(continues on next page)

(continued from previous page)

```

[-2.1079,  0.1999,  0.3236, -0.6416]],
[[-0.5875,  0.0539,  0.1949, -0.382 ],
 [ 0.3186,  2.0891, -0.7283, -0.0903],
 [-0.7482,  1.3189, -2.0298,  0.7927],
 [ 0.461 , -0.5427, -0.3054, -0.4792],
 [ 0.095 , -0.2701, -0.7071, -0.7739]]])

```

If a `DataFrame` or `Panel` contains homogeneously-typed data, the `ndarray` can actually be modified in-place, and the changes will be reflected in the data structure. For heterogeneous data (e.g. some of the `DataFrame`'s columns are not all the same dtype), this will not be the case. The values attribute itself, unlike the axis labels, cannot be assigned to.

**Note:** When working with heterogeneous data, the dtype of the resulting `ndarray` will be chosen to accommodate all of the data involved. For example, if strings are involved, the result will be of object dtype. If there are only floats and integers, the resulting array will be of float dtype.

## 9.3 Accelerated operations

pandas has support for accelerating certain types of binary numerical and boolean operations using the `numexpr` library and the `bottleneck` libraries.

These libraries are especially useful when dealing with large data sets, and provide large speedups. `numexpr` uses smart chunking, caching, and multiple cores. `bottleneck` is a set of specialized cython routines that are especially fast when dealing with arrays that have nans.

Here is a sample (using 100 column x 100,000 row `DataFrames`):

| Operation                 | 0.11.0 (ms) | Prior Version (ms) | Ratio to Prior |
|---------------------------|-------------|--------------------|----------------|
| <code>df1 &gt; df2</code> | 13.32       | 125.35             | 0.1063         |
| <code>df1 * df2</code>    | 21.71       | 36.63              | 0.5928         |
| <code>df1 + df2</code>    | 22.04       | 36.50              | 0.6039         |

You are highly encouraged to install both libraries. See the section [Recommended Dependencies](#) for more installation info.

These are both enabled to be used by default, you can control this by setting the options:

New in version 0.20.0.

```

pd.set_option('compute.use_bottleneck', False)
pd.set_option('compute.use_numexpr', False)

```

## 9.4 Flexible binary operations

With binary operations between pandas data structures, there are two key points of interest:

- Broadcasting behavior between higher- (e.g. `DataFrame`) and lower-dimensional (e.g. `Series`) objects.
- Missing data in computations.

We will demonstrate how to manage these issues independently, though they can be handled simultaneously.

### 9.4.1 Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, ... for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the *axis* keyword:

```
In [14]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c']
↳ ),
.....:                    'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c'
↳ ', 'd']),
.....:                    'three' : pd.Series(np.random.randn(3), index=['b', 'c',
↳ 'd'])})
.....:

In [15]: df
Out[15]:
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | -1.101558 | 1.124472  | NaN       |
| b | -0.177289 | 2.487104  | -0.634293 |
| c | 0.462215  | -0.486066 | 1.931194  |
| d | NaN       | -0.456288 | -1.222918 |

```
In [16]: row = df.iloc[1]

In [17]: column = df['two']

In [18]: df.sub(row, axis='columns')
Out[18]:
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | -0.924269 | -1.362632 | NaN       |
| b | 0.000000  | 0.000000  | 0.000000  |
| c | 0.639504  | -2.973170 | 2.565487  |
| d | NaN       | -2.943392 | -0.588625 |

```
In [19]: df.sub(row, axis=1)
////////////////////////////////////
↳
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | -0.924269 | -1.362632 | NaN       |
| b | 0.000000  | 0.000000  | 0.000000  |
| c | 0.639504  | -2.973170 | 2.565487  |
| d | NaN       | -2.943392 | -0.588625 |

```
In [20]: df.sub(column, axis='index')
////////////////////////////////////
↳
```

|   | one       | two | three     |
|---|-----------|-----|-----------|
| a | -2.226031 | 0.0 | NaN       |
| b | -2.664393 | 0.0 | -3.121397 |
| c | 0.948280  | 0.0 | 2.417260  |
| d | NaN       | 0.0 | -0.766631 |

```
In [21]: df.sub(column, axis=0)
////////////////////////////////////
↳
```

|   | one       | two | three     |
|---|-----------|-----|-----------|
| a | -2.226031 | 0.0 | NaN       |
| b | -2.664393 | 0.0 | -3.121397 |

(continues on next page)

|   |          |     |           |
|---|----------|-----|-----------|
| c | 0.948280 | 0.0 | 2.417260  |
| d | NaN      | 0.0 | -0.766631 |

```
In [22]: dfmi = df.copy()

In [23]: dfmi.index = pd.MultiIndex.from_tuples([(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a')],
.....:                                         names=['first', 'second'])
.....:

In [24]: dfmi.sub(column, axis=0, level='second')
Out[24]:
```

|       |        | one       | two      | three     |
|-------|--------|-----------|----------|-----------|
| first | second |           |          |           |
| 1     | a      | -2.226031 | 0.00000  | NaN       |
|       | b      | -2.664393 | 0.00000  | -3.121397 |
|       | c      | 0.948280  | 0.00000  | 2.417260  |
| 2     | a      | NaN       | -1.58076 | -2.347391 |

```
In [25]: major_mean = wp.mean(axis='major')

In [26]: major_mean
Out[26]:
```

|   | Item1     | Item2     |
|---|-----------|-----------|
| A | -0.878036 | -0.092218 |
| B | -0.060128 | 0.529811  |
| C | 0.099453  | -0.715139 |
| D | 0.248599  | -0.186535 |

```
In [27]: wp.sub(major_mean, axis='major')
\\////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↪
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

```
In [28]: s = pd.Series(np.arange(10))

In [29]: s
Out[29]:
```

(continued from previous page)

```

0      0
1      1
2      2
3      3
4      4
5      5
6      6
7      7
8      8
9      9
dtype: int64

In [30]: div, rem = divmod(s, 3)

In [31]: div
Out[31]:
0      0
1      0
2      0
3      1
4      1
5      1
6      2
7      2
8      2
9      3
dtype: int64

In [32]: rem
////////////////////////////////////Out[32]:
↪
0      0
1      1
2      2
3      0
4      1
5      2
6      0
7      1
8      2
9      0
dtype: int64

In [33]: idx = pd.Index(np.arange(10))

In [34]: idx
Out[34]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [35]: div, rem = divmod(idx, 3)

In [36]: div
Out[36]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [37]: rem
////////////////////////////////////Out[37]:
↪Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```



We can also do elementwise `divmod()`:

```
In [38]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

```
In [39]: div
```

```
Out[39]:
```

```
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int64
```

```
In [40]: rem
```

```
Out[40]:
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
9    3
dtype: int64
```

## 9.4.2 Missing data / operations with fill values

In Series and DataFrame, the arithmetic functions have the option of inputting a *fill\_value*, namely a value to substitute when at most one of the values at a location are missing. For example, when adding two DataFrame objects, you may wish to treat NaN as 0 unless both DataFrames are missing that value, in which case the result will be NaN (you can later replace NaN with some other value using `fillna` if you wish).

```
In [41]: df
```

```
Out[41]:
```

```
   one    two    three
a -1.101558  1.124472    NaN
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194
d      NaN -0.456288 -1.222918
```

```
In [42]: df2
```

```
   one    two    three
a -1.101558  1.124472  1.000000
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194
d      NaN -0.456288 -1.222918
```

(continues on next page)

(continued from previous page)

**In [43]:** df + df2

```

////////////////////////////////////
↪
      one      two      three
a -2.203116  2.248945      NaN
b -0.354579  4.974208 -1.268586
c  0.924429 -0.972131  3.862388
d         NaN -0.912575 -2.445837

```

**In [44]:** df.add(df2, fill\_value=0)

```

////////////////////////////////////
↪
      one      two      three
a -2.203116  2.248945  1.000000
b -0.354579  4.974208 -1.268586
c  0.924429 -0.972131  3.862388
d         NaN -0.912575 -2.445837

```

### 9.4.3 Flexible Comparisons

Series and DataFrame have the binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` whose behavior is analogous to the binary arithmetic operations described above:

**In [45]:** df.gt(df2)**Out[45]:**

```

      one      two      three
a False False False
b False False False
c False False False
d False False False

```

**In [46]:** df2.ne(df)

```

////////////////////////////////////
↪
      one      two      three
a False False  True
b False False False
c False False False
d  True False False

```

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype `bool`. These boolean objects can be used in indexing operations, see the section on [Boolean indexing](#).

### 9.4.4 Boolean Reductions

You can apply the reductions: `empty()`, `any()`, `all()`, and `bool()` to provide a way to summarize a boolean result.

**In [47]:** (df > 0).all()**Out[47]:**

```

one      False
two      False
three    False

```

(continues on next page)

(continued from previous page)

```
dtype: bool

In [48]: (df > 0).any()
\\Out[48]:
one      True
two      True
three    True
dtype: bool
```

You can reduce to a final boolean value.

```
In [49]: (df > 0).any().any()
Out[49]: True
```

You can test if a pandas object is empty, via the *empty* property.

```
In [50]: df.empty
Out[50]: False

In [51]: pd.DataFrame(columns=list('ABC')).empty
\\Out[51]: True
```

To evaluate single-element pandas objects in a boolean context, use the method *bool()*:

```
In [52]: pd.Series([True]).bool()
Out[52]: True

In [53]: pd.Series([False]).bool()
\\Out[53]: False

In [54]: pd.DataFrame([[True]]).bool()
\\Out[54]: True

In [55]: pd.DataFrame([[False]]).bool()
\\Out[55]: False
```

**Warning:** You might be tempted to do the following:

```
>>> if df:
    ...
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.

```
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.
->all().
```

See *gotchas* for a more detailed discussion.

### 9.4.5 Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider `df+df` and `df*2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df+df == df*2).all()`. But in fact, this expression is `False`:

```
In [56]: df+df == df*2
```

```
Out[56]:
```

```
      one  two  three
a   True  True  False
b   True  True   True
c   True  True   True
d  False  True   True
```

```
In [57]: (df+df == df*2).all()
```

```

////////////////////////////////////
↪
one      False
two       True
three    False
dtype: bool
```

Notice that the boolean DataFrame `df+df == df*2` contains some `False` values! This is because NaNs do not compare as equals:

```
In [58]: np.nan == np.nan
```

```
Out[58]: False
```

So, NDFrames (such as Series, DataFrames, and Panels) have an `equals()` method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [59]: (df+df).equals(df*2)
```

```
Out[59]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be `True`:

```
In [60]: df1 = pd.DataFrame({'col': ['foo', 0, np.nan]})
```

```
In [61]: df2 = pd.DataFrame({'col': [np.nan, 0, 'foo']}, index=[2,1,0])
```

```
In [62]: df1.equals(df2)
```

```
Out[62]: False
```

```
In [63]: df1.equals(df2.sort_index())
```

```
Out[63]: True
```

### 9.4.6 Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [64]: pd.Series(['foo', 'bar', 'baz']) == 'foo'
```

```
Out[64]:
```

```
0      True
1     False
2     False
dtype: bool
```

(continues on next page)

(continued from previous page)

```
In [65]: pd.Index(['foo', 'bar', 'baz']) == 'foo'
Out[65]: array([ True, False,  False], dtype=bool)
```

Pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [66]: pd.Series(['foo', 'bar', 'baz']) == pd.Index(['foo', 'bar', 'qux'])
Out[66]:
0    True
1    True
2   False
dtype: bool

In [67]: pd.Series(['foo', 'bar', 'baz']) == np.array(['foo', 'bar', 'qux'])
Out[67]:
0    True
1    True
2   False
dtype: bool
```

Trying to compare Index or Series objects of different lengths will raise a `ValueError`:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare

In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [68]: np.array([1, 2, 3]) == np.array([2])
Out[68]: array([False,  True, False], dtype=bool)
```

or it can return `False` if broadcasting can not be done:

```
In [69]: np.array([1, 2, 3]) == np.array([1, 2])
Out[69]: False
```

## 9.4.7 Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of “higher quality”. However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two `DataFrame` objects where missing values in one `DataFrame` are conditionally filled with like-labeled values from the other `DataFrame`. The function implementing this operation is `combine_first()`, which we illustrate:

```
In [70]: df1 = pd.DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
.....:                      'B' : [np.nan, 2., 3., np.nan, 6.]})

In [71]: df2 = pd.DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
.....:                      'B' : [np.nan, np.nan, 3., 4., 6., 8.]})
```

(continues on next page)

(continued from previous page)

```

.....:

In [72]: df1
Out[72]:
   A    B
0  1.0 NaN
1  NaN  2.0
2  3.0  3.0
3  5.0 NaN
4  NaN  6.0

In [73]: df2
Out[73]:
   A    B
0  5.0 NaN
1  2.0 NaN
2  4.0  3.0
3  NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [74]: df1.combine_first(df2)
Out[74]:
   A    B
0  1.0 NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0

```

### 9.4.8 General DataFrame Combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```

In [75]: combiner = lambda x, y: np.where(pd.isna(x), y, x)

In [76]: df1.combine(df2, combiner)
Out[76]:
   A    B
0  1.0 NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0

```

## 9.5 Descriptive statistics

There exists a large number of methods for computing descriptive statistics and other related operations on *Series*, *DataFrame*, and *Panel*. Most of these are aggregations (hence producing a lower-dimensional result) like `sum()`, `mean()`, and `quantile()`, but some of them, like `cumsum()` and `cumprod()`, produce an object of the same size. Generally speaking, these methods take an **axis** argument, just like `ndarray.{sum, std, ...}`, but the axis can be specified by name or integer:

- **Series:** no axis argument needed
- **DataFrame:** “index” (axis=0, default), “columns” (axis=1)
- **Panel:** “items” (axis=0), “major” (axis=1, default), “minor” (axis=2)

For example:

```
In [77]: df
Out[77]:
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | -1.101558 | 1.124472  | NaN       |
| b | -0.177289 | 2.487104  | -0.634293 |
| c | 0.462215  | -0.486066 | 1.931194  |
| d | NaN       | -0.456288 | -1.222918 |

```
In [78]: df.mean(0)
```

```

////////////////////////////////////
↪
one      -0.272211
two       0.667306
three     0.024661
dtype: float64
```

```
In [79]: df.mean(1)
```

```

////////////////////////////////////
↪
a      0.011457
b      0.558507
c      0.635781
d     -0.839603
dtype: float64
```

All such methods have a `skipna` option signaling whether to exclude missing data (True by default):

```
In [80]: df.sum(0, skipna=False)
```

```

Out[80]:
one      NaN
two      2.669223
three     NaN
dtype: float64
```

```
In [81]: df.sum(axis=1, skipna=True)
```

```

////////////////////////////////////Out[81]:
↪
a      0.022914
b      1.675522
c      1.907343
d     -1.679206
dtype: float64
```

Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```
In [82]: ts_stand = (df - df.mean()) / df.std()

In [83]: ts_stand.std()
Out[83]:
one      1.0
two      1.0
three    1.0
dtype: float64

In [84]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)

In [85]: xs_stand.std(1)
Out[85]:
a      1.0
b      1.0
c      1.0
d      1.0
dtype: float64
```

Note that methods like `cumsum()` and `cumprod()` preserve the location of NaN values. This is somewhat different from `expanding()` and `rolling()`. For more details please see [this note](#).

```
In [86]: df.cumsum()
Out[86]:
```

|   | one       | two      | three     |
|---|-----------|----------|-----------|
| a | -1.101558 | 1.124472 | NaN       |
| b | -1.278848 | 3.611576 | -0.634293 |
| c | -0.816633 | 3.125511 | 1.296901  |
| d | NaN       | 2.669223 | 0.073983  |

Here is a quick reference summary table of common functions. Each also takes an optional `level` parameter which applies only if the object has a [hierarchical index](#).



| Function | Description                                |
|----------|--|
| count    | Number of non-NA observations              |
| sum      | Sum of values                              |
| mean     | Mean of values                             |
| mad      | Mean absolute deviation                    |
| median   | Arithmetic median of values                |
| min      | Minimum                                    |
| max      | Maximum                                    |
| mode     | Mode                                       |
| abs      | Absolute Value                             |
| prod     | Product of values                          |
| std      | Bessel-corrected sample standard deviation |
| var      | Unbiased variance                          |
| sem      | Standard error of the mean                 |
| skew     | Sample skewness (3rd moment)               |
| kurt     | Sample kurtosis (4th moment)               |
| quantile | Sample quantile (value at %)               |
| cumsum   | Cumulative sum                             |
| cumprod  | Cumulative product                         |
| cummax   | Cumulative maximum                         |
| cummin   | Cumulative minimum                         |

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```
In [87]: np.mean(df['one'])
Out[87]: -0.27221094480450114

In [88]: np.mean(df['one'].values)
Out[88]: nan
```

`Series.nunique()` will return the number of unique non-NA values in a Series:

```
In [89]: series = pd.Series(np.random.randn(500))

In [90]: series[20:500] = np.nan

In [91]: series[10:20] = 5

In [92]: series.nunique()
Out[92]: 11
```

### 9.5.1 Summarizing data: describe

There is a convenient `describe()` function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```
In [93]: series = pd.Series(np.random.randn(1000))

In [94]: series[::2] = np.nan

In [95]: series.describe()
Out[95]:
```

(continues on next page)

(continued from previous page)

```

count      500.000000
mean       -0.032127
std        1.067484
min        -3.463789
25%        -0.725523
50%        -0.053230
75%         0.679790
max         3.120271
dtype: float64

In [96]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd',
↳ 'e'])

In [97]: frame.iloc[::2] = np.nan

In [98]: frame.describe()
Out[98]:
```

|       | a          | b          | c          | d          | e          |
|-------|------------|------------|------------|------------|------------|
| count | 500.000000 | 500.000000 | 500.000000 | 500.000000 | 500.000000 |
| mean  | -0.045109  | -0.052045  | 0.024520   | 0.006117   | 0.001141   |
| std   | 1.029268   | 1.002320   | 1.042793   | 1.040134   | 1.005207   |
| min   | -2.915767  | -3.294023  | -3.610499  | -2.907036  | -3.010899  |
| 25%   | -0.763783  | -0.720389  | -0.609600  | -0.665896  | -0.682900  |
| 50%   | -0.086033  | -0.048843  | 0.006093   | 0.043191   | -0.001651  |
| 75%   | 0.663399   | 0.620980   | 0.728382   | 0.735973   | 0.656439   |
| max   | 3.400646   | 2.925597   | 3.416896   | 3.331522   | 3.007143   |

You can select specific percentiles to include in the output:

```

In [99]: series.describe(percentiles=[.05, .25, .75, .95])
Out[99]:
```

|       |            |
|-------|------------|
| count | 500.000000 |
| mean  | -0.032127  |
| std   | 1.067484   |
| min   | -3.463789  |
| 5%    | -1.733545  |
| 25%   | -0.725523  |
| 50%   | -0.053230  |
| 75%   | 0.679790   |
| 95%   | 1.854383   |
| max   | 3.120271   |

```

dtype: float64

```

By default, the median is always included.

For a non-numerical Series object, `describe()` will give a simple summary of the number of unique values and most frequently occurring values:

```

In [100]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [101]: s.describe()
Out[101]:
```

|        |   |
|--------|---|
| count  | 9 |
| unique | 4 |
| top    | a |
| freq   | 5 |

```

dtype: object

```

Note that on a mixed-type DataFrame object, `describe()` will restrict the summary to include only numerical columns or, if none are, only categorical columns:

```
In [102]: frame = pd.DataFrame({'a': ['Yes', 'Yes', 'No', 'No'], 'b': range(4)})

In [103]: frame.describe()
Out[103]:
```

|       | b        |
|-------|----------|
| count | 4.000000 |
| mean  | 1.500000 |
| std   | 1.290994 |
| min   | 0.000000 |
| 25%   | 0.750000 |
| 50%   | 1.500000 |
| 75%   | 2.250000 |
| max   | 3.000000 |

This behaviour can be controlled by providing a list of types as `include/exclude` arguments. The special value `all` can also be used:

```
In [104]: frame.describe(include=['object'])
Out[104]:
```

|        | a   |
|--------|-----|
| count  | 4   |
| unique | 2   |
| top    | Yes |
| freq   | 2   |

```
In [105]: frame.describe(include=['number'])
Out[105]:
```

|       | b        |
|-------|----------|
| count | 4.000000 |
| mean  | 1.500000 |
| std   | 1.290994 |
| min   | 0.000000 |
| 25%   | 0.750000 |
| 50%   | 1.500000 |
| 75%   | 2.250000 |
| max   | 3.000000 |

```
In [106]: frame.describe(include='all')
Out[106]:
```

|        | a   | b        |
|--------|-----|----------|
| count  | 4   | 4.000000 |
| unique | 2   | NaN      |
| top    | Yes | NaN      |
| freq   | 2   | NaN      |
| mean   | NaN | 1.500000 |
| std    | NaN | 1.290994 |
| min    | NaN | 0.000000 |
| 25%    | NaN | 0.750000 |
| 50%    | NaN | 1.500000 |
| 75%    | NaN | 2.250000 |
| max    | NaN | 3.000000 |

That feature relies on `select_dtypes`. Refer to there for details about accepted inputs.

## 9.5.2 Index of Min/Max Values

The `idxmin()` and `idxmax()` functions on Series and DataFrame compute the index labels with the minimum and maximum corresponding values:

```
In [107]: s1 = pd.Series(np.random.randn(5))

In [108]: s1
Out[108]:
0    -1.649461
1     0.169660
2     1.246181
3     0.131682
4    -2.001988
dtype: float64

In [109]: s1.idxmin(), s1.idxmax()
Out[109]:
(4, 2)

In [110]: df1 = pd.DataFrame(np.random.randn(5, 3), columns=['A', 'B', 'C'])

In [111]: df1
Out[111]:
   A         B         C
0 -1.273023  0.870502  0.214583
1  0.088452 -0.173364  1.207466
2  0.546121  0.409515 -0.310515
3  0.585014 -0.490528 -0.054639
4 -0.239226  0.701089  0.228656

In [112]: df1.idxmin(axis=0)
Out[112]:
A    0
B    3
C    2
dtype: int64

In [113]: df1.idxmax(axis=1)
Out[113]:
0    B
1    C
2    A
3    A
4    B
dtype: object
```

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin()` and `idxmax()` return the first matching index:

```
In [114]: df3 = pd.DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [115]: df3
Out[115]:
   A
e  2.0
```

(continues on next page)

```
In [116]: df3['A'].idxmin()
Out[116]: 'd'
```

```
In [118]: data
Out[118]:
array([[3, 3, 0, 2, 1, 0, 5, 5, 3, 6, 1, 5, 6, 2, 0, 0, 6, 3, 3, 5, 0, 4, 3,
        3, 3, 0, 6, 1, 3, 5, 5, 0, 4, 0, 6, 3, 6, 5, 4, 3, 2, 1, 5, 0, 1, 1,
        6, 4, 1, 4]])
```

```
In [120]: s.value_counts()
```

```
3      11
0       9
5       8
6       7
1       7
4       5
2       3
dtype: int64
```

```
Out[121]:
3      11
0       9
5       8
6       7
1       7
4       5
2       3
dtype: int64
```

```
In [123]: s5.mode()
```

(continued from previous page)

```

Out[123]:
0      3
1      7
dtype: int64

In [124]: df5 = pd.DataFrame({"A": np.random.randint(0, 7, size=50),
.....:                      "B": np.random.randint(-10, 15, size=50)})
.....:

In [125]: df5.mode()
Out[125]:
   A  B
0  2 -5

```

## 9.5.4 Discretization and quantiling

Continuous values can be discretized using the `cut()` (bins based on values) and `qcut()` (bins based on sample quantiles) functions:

```

In [126]: arr = np.random.randn(20)

In [127]: factor = pd.cut(arr, 4)

In [128]: factor
Out[128]:
[(-2.611, -1.58], (0.473, 1.499], (-2.611, -1.58], (-1.58, -0.554], (-0.554, 0.473], .
↪.., (0.473, 1.499], (0.473, 1.499], (-0.554, 0.473], (-0.554, 0.473], (-0.554, 0.
↪473]]
Length: 20
Categories (4, interval[float64]): [(-2.611, -1.58] < (-1.58, -0.554] < (-0.554, 0.
↪473] <
                                     (0.473, 1.499]]

In [129]: factor = pd.cut(arr, [-5, -1, 0, 1, 5])

In [130]: factor
Out[130]:
[(-5, -1], (0, 1], (-5, -1], (-1, 0], (-1, 0], ..., (1, 5], (1, 5], (-1, 0], (-1, 0], ↪
↪(-1, 0]]
Length: 20
Categories (4, interval[int64]): [(-5, -1] < (-1, 0] < (0, 1] < (1, 5]]

```

`qcut()` computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quantiles like so:

```

In [131]: arr = np.random.randn(30)

In [132]: factor = pd.qcut(arr, [0, .25, .5, .75, 1])

In [133]: factor
Out[133]:
[(0.544, 1.976], (0.544, 1.976], (-1.255, -0.375], (0.544, 1.976], (-0.103, 0.544], ..
↪.., (-0.103, 0.544], (0.544, 1.976], (-0.103, 0.544], (-1.255, -0.375], (-0.375, -0.
↪103]]
Length: 30

```

(continues on next page)

(continued from previous page)

```
Categories (4, interval[float64]): [(-1.255, -0.375] < (-0.375, -0.103] < (-0.103, 0.
↪ 544] <
                                     (0.544, 1.976]]
```

```
In [134]: pd.value_counts(factor)
```

```
//////////////////////////////////////
↪
(0.544, 1.976]      8
(-1.255, -0.375]   8
(-0.103, 0.544]    7
(-0.375, -0.103]   7
dtype: int64
```

We can also pass infinite values to define the bins:

```
In [135]: arr = np.random.randn(20)
```

```
In [136]: factor = pd.cut(arr, [-np.inf, 0, np.inf])
```

```
In [137]: factor
```

```
Out[137]:
[(0.0, inf], (0.0, inf], (0.0, inf], (0.0, inf], (-inf, 0.0], ..., (-inf, 0.0], (-inf,
↪ 0.0], (0.0, inf], (-inf, 0.0], (0.0, inf]]
Length: 20
Categories (2, interval[float64]): [(-inf, 0.0] < (0.0, inf]]
```

## 9.6 Function application

To apply your own or another library's functions to pandas objects, you should be aware of the three methods below. The appropriate method to use depends on whether your function expects to operate on an entire `DataFrame` or `Series`, row- or column-wise, or elementwise.

1. *Tablewise Function Application: `pipe()`*
2. *Row or Column-wise Function Application: `apply()`*
3. *Aggregation API: `agg()` and `transform()`*
4. *Applying Elementwise Functions: `applymap()`*

### 9.6.1 Tablewise Function Application

`DataFrames` and `Series` can of course just be passed into functions. However, if the function needs to be called in a chain, consider using the `pipe()` method. Compare the following

```
# f, g, and h are functions taking and returning ``DataFrames``
>>> f(g(h(df), arg1=1), arg2=2, arg3=3)
```

with the equivalent

```
>>> (df.pipe(h)
     .pipe(g, arg1=1)
     .pipe(f, arg2=2, arg3=3)
     )
```

Pandas encourages the second style, which is known as method chaining. `pipe` makes it easy to use your own or another library's functions in method chains, alongside pandas' methods.

In the example above, the functions `f`, `g`, and `h` each expected the `DataFrame` as the first positional argument. What if the function you wish to apply takes its data as, say, the second argument? In this case, provide `pipe` with a tuple of (callable, `data_keyword`). `pipe` will route the `DataFrame` to the argument specified in the tuple.

For example, we can fit a regression using `statsmodels`. Their API expects a formula first and a `DataFrame` as the second argument, `data`. We pass in the function, keyword pair (`sm.ols`, `'data'`) to `pipe`:

```
In [138]: import statsmodels.formula.api as sm

In [139]: bb = pd.read_csv('data/baseball.csv', index_col='id')

In [140]: (bb.query('h > 0')
.....:      .assign(ln_h = lambda df: np.log(df.h))
.....:      .pipe((sm.ols, 'data'), 'hr ~ ln_h + year + g + C(lg)')
.....:      .fit()
.....:      .summary()
.....: )
Out[140]:
<class 'statsmodels.iolib.summary.Summary'>
"""
                        OLS Regression Results
=====
Dep. Variable:          hr      R-squared:                0.685
Model:                  OLS      Adj. R-squared:           0.665
Method:                 Least Squares      F-statistic:        34.28
Date:                   Thu, 05 Jul 2018    Prob (F-statistic):    3.48e-15
Time:                   17:07:05           Log-Likelihood:     -205.92
No. Observations:       68              AIC:                  421.8
Df Residuals:           63              BIC:                  432.9
Df Model:                4
Covariance Type:        nonrobust
=====
                        coef      std err          t      P>|t|      [0.025      0.975]
-----
Intercept      -8484.7720     4664.146     -1.819     0.074    -1.78e+04     835.780
C(lg) [T.NL]    -2.2736         1.325     -1.716     0.091     -4.922         0.375
ln_h            -1.3542         0.875     -1.547     0.127     -3.103         0.395
year             4.2277         2.324         1.819     0.074     -0.417         8.872
g                0.1841         0.029         6.258     0.000         0.125         0.243
=====
Omnibus:                 10.875    Durbin-Watson:           1.999
Prob(Omnibus):           0.004    Jarque-Bera (JB):         17.298
Skew:                    0.537    Prob(JB):                 0.000175
Kurtosis:                5.225    Cond. No.:                1.49e+07
=====

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly_
    ↪specified.
[2] The condition number is large, 1.49e+07. This might indicate that there are
    strong multicollinearity or other numerical problems.
"""
```

The `pipe` method is inspired by unix pipes and more recently `dplyr` and `magrittr`, which have introduced the popular (`%>%`) (read pipe) operator for `R`. The implementation of `pipe` here is quite clean and feels right at home in python.



We encourage you to view the source code of `pipe()`.

## 9.6.2 Row or Column-wise Function Application

Arbitrary functions can be applied along the axes of a DataFrame using the `apply()` method, which, like the descriptive statistics methods, takes an optional `axis` argument:

```
In [141]: df.apply(np.mean)
Out[141]:
one      -0.272211
two       0.667306
three     0.024661
dtype: float64

In [142]: df.apply(np.mean, axis=1)
Out[142]:
a      0.011457
b      0.558507
c      0.635781
d     -0.839603
dtype: float64

In [143]: df.apply(lambda x: x.max() - x.min())
Out[143]:
one      1.563773
two      2.973170
three    3.154112
dtype: float64

In [144]: df.apply(np.cumsum)
Out[144]:
      one      two      three
a -1.101558  1.124472      NaN
b -1.278848  3.611576 -0.634293
c -0.816633  3.125511  1.296901
d      NaN  2.669223  0.073983

In [145]: df.apply(np.exp)
Out[145]:
      one      two      three
a  0.332353  3.078592      NaN
b  0.837537 12.026397  0.53031
c  1.587586  0.615041  6.89774
d      NaN  0.633631  0.29437
```

The `apply()` method will also dispatch on a string method name.

```
In [146]: df.apply('mean')
Out[146]:
one      -0.272211
two       0.667306
three     0.024661
dtype: float64
```

(continues on next page)

(continued from previous page)

```

In [147]: df.apply('mean', axis=1)
Out[147]:
a    0.011457
b    0.558507
c    0.635781
d   -0.839603
dtype: float64

```

The return type of the function passed to `apply()` affects the type of the final output from `DataFrame.apply` for the default behaviour:

- If the applied function returns a `Series`, the final output is a `DataFrame`. The columns match the index of the `Series` returned by the applied function.
- If the applied function returns any other type, the final output is a `Series`.

This default behaviour can be overridden using the `result_type`, which accepts three options: `reduce`, `broadcast`, and `expand`. These will determine how list-like return values expand (or not) to a `DataFrame`.

`apply()` combined with some cleverness can be used to answer many questions about a data set. For example, suppose we wanted to extract the date where the maximum value for each column occurred:

```

In [148]: tsdf = pd.DataFrame(np.random.randn(1000, 3), columns=['A', 'B', 'C'],
.....:                        index=pd.date_range('1/1/2000', periods=1000))
.....:

In [149]: tsdf.apply(lambda x: x.idxmax())
Out[149]:
A    2001-04-25
B    2002-05-31
C    2002-09-25
dtype: datetime64[ns]

```

You may also pass additional arguments and keyword arguments to the `apply()` method. For instance, consider the following function you would like to apply:

```

def subtract_and_divide(x, sub, divide=1):
    return (x - sub) / divide

```

You may then apply this function as follows:

```
df.apply(subtract_and_divide, args=(5,), divide=3)
```

Another useful feature is the ability to pass `Series` methods to carry out some `Series` operation on each column or row:

```

In [150]: tsdf
Out[150]:
          A          B          C
2000-01-01 -0.720299  0.546303 -0.082042
2000-01-02  0.200295 -0.577554 -0.908402
2000-01-03  0.102533  1.653614  0.303319
2000-01-04         NaN         NaN         NaN
2000-01-05         NaN         NaN         NaN
2000-01-06         NaN         NaN         NaN
2000-01-07         NaN         NaN         NaN

```

(continues on next page)

(continued from previous page)

```

2000-01-08  0.532566  0.341548  0.150493
2000-01-09  0.330418  1.761200  0.567133
2000-01-10 -0.251020  1.020099  1.893177

```

```
In [151]: tsdf.apply(pd.Series.interpolate)
```

```

////////////////////////////////////
↪
      A      B      C
2000-01-01 -0.720299  0.546303 -0.082042
2000-01-02  0.200295 -0.577554 -0.908402
2000-01-03  0.102533  1.653614  0.303319
2000-01-04  0.188539  1.391201  0.272754
2000-01-05  0.274546  1.128788  0.242189
2000-01-06  0.360553  0.866374  0.211624
2000-01-07  0.446559  0.603961  0.181059
2000-01-08  0.532566  0.341548  0.150493
2000-01-09  0.330418  1.761200  0.567133
2000-01-10 -0.251020  1.020099  1.893177

```

Finally, `apply()` takes an argument `raw` which is `False` by default, which converts each row or column into a `Series` before applying the function. When set to `True`, the passed function will instead receive an `ndarray` object, which has positive performance implications if you do not need the indexing functionality.

### 9.6.3 Aggregation API

New in version 0.20.0.

The aggregation API allows one to express possibly multiple aggregation operations in a single concise way. This API is similar across pandas objects, see [groupby API](#), the [window functions API](#), and the [resample API](#). The entry point for aggregation is `DataFrame.aggregate()`, or the alias `DataFrame.agg()`.

We will use a similar starting frame from above:

```
In [152]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
.....:                        index=pd.date_range('1/1/2000', periods=10))
.....:
```

```
In [153]: tsdf.iloc[3:7] = np.nan
```

```
In [154]: tsdf
```

```

Out[154]:
      A      B      C
2000-01-01  0.170247 -0.916844  0.835024
2000-01-02  1.259919  0.801111  0.445614
2000-01-03  1.453046  2.430373  0.653093
2000-01-04      NaN      NaN      NaN
2000-01-05      NaN      NaN      NaN
2000-01-06      NaN      NaN      NaN
2000-01-07      NaN      NaN      NaN
2000-01-08 -1.874526  0.569822 -0.609644
2000-01-09  0.812462  0.565894 -1.461363
2000-01-10 -0.985475  1.388154 -0.078747

```

Using a single function is equivalent to `apply()`. You can also pass named methods as strings. These will return a `Series` of the aggregated output:

```

In [155]: tsdf.agg(np.sum)
Out[155]:
A    0.835673
B    4.838510
C   -0.216025
dtype: float64

In [156]: tsdf.agg('sum')
Out[156]:
A    0.835673
B    4.838510
C   -0.216025
dtype: float64

# these are equivalent to a ``.sum()`` because we are aggregating on a single function
In [157]: tsdf.sum()
Out[157]:
A    0.835673
B    4.838510
C   -0.216025
dtype: float64

```

Single aggregations on a Series this will return a scalar value:

```

In [158]: tsdf.A.agg('sum')
Out[158]: 0.83567297915820504

```

### 9.6.3.1 Aggregating with multiple functions

You can pass multiple aggregation arguments as a list. The results of each of the passed functions will be a row in the resulting DataFrame. These are naturally named from the aggregation function.

```

In [159]: tsdf.agg(['sum'])
Out[159]:
      A      B      C
sum  0.835673  4.83851 -0.216025

```

Multiple functions yield multiple rows:

```

In [160]: tsdf.agg(['sum', 'mean'])
Out[160]:
      A      B      C
sum  0.835673  4.838510 -0.216025
mean 0.139279  0.806418 -0.036004

```

On a Series, multiple functions return a Series, indexed by the function names:

```

In [161]: tsdf.A.agg(['sum', 'mean'])
Out[161]:
sum    0.835673
mean   0.139279
Name: A, dtype: float64

```

Passing a lambda function will yield a <lambda> named row:

```
In [162]: tsdf.A.agg(['sum', lambda x: x.mean()])
Out[162]:
sum          0.835673
<lambda>     0.139279
Name: A, dtype: float64
```

Passing a named function will yield that name for the row:

```
In [163]: def mymean(x):
.....:     return x.mean()
.....:

In [164]: tsdf.A.agg(['sum', mymean])
Out[164]:
sum          0.835673
mymean       0.139279
Name: A, dtype: float64
```

### 9.6.3.2 Aggregating with a dict

Passing a dictionary of column names to a scalar or a list of scalars, to `DataFrame.agg` allows you to customize which functions are applied to which columns. Note that the results are not in any particular order, you can use an `OrderedDict` instead to guarantee ordering.

```
In [165]: tsdf.agg({'A': 'mean', 'B': 'sum'})
Out[165]:
A      0.139279
B      4.838510
dtype: float64
```

Passing a list-like will generate a `DataFrame` output. You will get a matrix-like output of all of the aggregators. The output will consist of all unique functions. Those that are not noted for a particular column will be `NaN`:

```
In [166]: tsdf.agg({'A': ['mean', 'min'], 'B': 'sum'})
Out[166]:
      A      B
mean  0.139279  NaN
min   -1.874526  NaN
sum      NaN  4.83851
```

### 9.6.3.3 Mixed Dtypes

When presented with mixed dtypes that cannot aggregate, `.agg` will only take the valid aggregations. This is similar to how `groupby.agg` works.

```
In [167]: mdf = pd.DataFrame({'A': [1, 2, 3],
.....:                        'B': [1., 2., 3.],
.....:                        'C': ['foo', 'bar', 'baz'],
.....:                        'D': pd.date_range('20130101', periods=3)})
.....:

In [168]: mdf.dtypes
Out[168]:
A      int64
```

(continues on next page)

(continued from previous page)

```

B          float64
C          object
D    datetime64[ns]
dtype: object

```

```

In [169]: mdf.agg(['min', 'sum'])
Out[169]:
      A      B      C      D
min  1  1.0      bar 2013-01-01
sum  6  6.0 foobarbaz      NaT

```

### 9.6.3.4 Custom describe

With `.agg()` it is possible to easily create a custom describe function, similar to the built in *describe* function.

```

In [170]: from functools import partial

In [171]: q_25 = partial(pd.Series.quantile, q=0.25)

In [172]: q_25.__name__ = '25%'

In [173]: q_75 = partial(pd.Series.quantile, q=0.75)

In [174]: q_75.__name__ = '75%'

In [175]: tsdf.agg(['count', 'mean', 'std', 'min', q_25, 'median', q_75, 'max'])
Out[175]:
      A      B      C
count  6.000000  6.000000  6.000000
mean   0.139279  0.806418 -0.036004
std    1.323362  1.100830  0.874990
min   -1.874526 -0.916844 -1.461363
25%   -0.696544  0.566876 -0.476920
median  0.491354  0.685467  0.183433
75%    1.148055  1.241393  0.601223
max    1.453046  2.430373  0.835024

```

### 9.6.4 Transform API

New in version 0.20.0.

The *transform()* method returns an object that is indexed the same (same size) as the original. This API allows you to provide *multiple* operations at the same time rather than one-by-one. Its API is quite similar to the `.agg` API.

We create a frame similar to the one used in the above sections.

```

In [176]: tsdf = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
.....:                        index=pd.date_range('1/1/2000', periods=10))
.....:

In [177]: tsdf.iloc[3:7] = np.nan

In [178]: tsdf
Out[178]:

```

(continues on next page)

|            | A         | B         | C         |
|------------|-----------|-----------|-----------|
| 2000-01-01 | -0.578465 | -0.503335 | -0.987140 |
| 2000-01-02 | -0.767147 | -0.266046 | 1.083797  |
| 2000-01-03 | 0.195348  | 0.722247  | -0.894537 |
| 2000-01-04 | NaN       | NaN       | NaN       |
| 2000-01-05 | NaN       | NaN       | NaN       |
| 2000-01-06 | NaN       | NaN       | NaN       |
| 2000-01-07 | NaN       | NaN       | NaN       |
| 2000-01-08 | -0.556397 | 0.542165  | -0.308675 |
| 2000-01-09 | -1.010924 | -0.672504 | -1.139222 |
| 2000-01-10 | 0.354653  | 0.563622  | -0.365106 |

Out [179]:

|            | A        | B        | C        |
|------------|----------|----------|----------|
| 2000-01-01 | 0.578465 | 0.503335 | 0.987140 |
| 2000-01-02 | 0.767147 | 0.266046 | 1.083797 |
| 2000-01-03 | 0.195348 | 0.722247 | 0.894537 |
| 2000-01-04 | NaN      | NaN      | NaN      |
| 2000-01-05 | NaN      | NaN      | NaN      |
| 2000-01-06 | NaN      | NaN      | NaN      |
| 2000-01-07 | NaN      | NaN      | NaN      |
| 2000-01-08 | 0.556397 | 0.542165 | 0.308675 |
| 2000-01-09 | 1.010924 | 0.672504 | 1.139222 |
| 2000-01-10 | 0.354653 | 0.563622 | 0.365106 |

[illegible]

(continued from previous page)

|            |          |          |          |
|------------|----------|----------|----------|
| 2000-01-09 | 1.010924 | 0.672504 | 1.139222 |
| 2000-01-10 | 0.354653 | 0.563622 | 0.365106 |

Here `transform()` received a single function; this is equivalent to a `ufunc` application.

```
In [182]: np.abs(tsdf)
Out[182]:
```

|            | A        | B        | C        |
|------------|----------|----------|----------|
| 2000-01-01 | 0.578465 | 0.503335 | 0.987140 |
| 2000-01-02 | 0.767147 | 0.266046 | 1.083797 |
| 2000-01-03 | 0.195348 | 0.722247 | 0.894537 |
| 2000-01-04 | NaN      | NaN      | NaN      |
| 2000-01-05 | NaN      | NaN      | NaN      |
| 2000-01-06 | NaN      | NaN      | NaN      |
| 2000-01-07 | NaN      | NaN      | NaN      |
| 2000-01-08 | 0.556397 | 0.542165 | 0.308675 |
| 2000-01-09 | 1.010924 | 0.672504 | 1.139222 |
| 2000-01-10 | 0.354653 | 0.563622 | 0.365106 |

Passing a single function to `.transform()` with a `Series` will yield a single `Series` in return.

```
In [183]: tsdf.A.transform(np.abs)
Out[183]:
```

|            |          |
|------------|----------|
| 2000-01-01 | 0.578465 |
| 2000-01-02 | 0.767147 |
| 2000-01-03 | 0.195348 |
| 2000-01-04 | NaN      |
| 2000-01-05 | NaN      |
| 2000-01-06 | NaN      |
| 2000-01-07 | NaN      |
| 2000-01-08 | 0.556397 |
| 2000-01-09 | 1.010924 |
| 2000-01-10 | 0.354653 |

Freq: D, Name: A, dtype: float64

### 9.6.4.1 Transform with multiple functions

Passing multiple functions will yield a column multi-indexed `DataFrame`. The first level will be the original frame column names; the second level will be the names of the transforming functions.

```
In [184]: tsdf.transform([np.abs, lambda x: x+1])
Out[184]:
```

|            | A        |           | B        |          | C        |           |
|------------|----------|-----------|----------|----------|----------|-----------|
|            | absolute | <lambda>  | absolute | <lambda> | absolute | <lambda>  |
| 2000-01-01 | 0.578465 | 0.421535  | 0.503335 | 0.496665 | 0.987140 | 0.012860  |
| 2000-01-02 | 0.767147 | 0.232853  | 0.266046 | 0.733954 | 1.083797 | 2.083797  |
| 2000-01-03 | 0.195348 | 1.195348  | 0.722247 | 1.722247 | 0.894537 | 0.105463  |
| 2000-01-04 | NaN      | NaN       | NaN      | NaN      | NaN      | NaN       |
| 2000-01-05 | NaN      | NaN       | NaN      | NaN      | NaN      | NaN       |
| 2000-01-06 | NaN      | NaN       | NaN      | NaN      | NaN      | NaN       |
| 2000-01-07 | NaN      | NaN       | NaN      | NaN      | NaN      | NaN       |
| 2000-01-08 | 0.556397 | 0.443603  | 0.542165 | 1.542165 | 0.308675 | 0.691325  |
| 2000-01-09 | 1.010924 | -0.010924 | 0.672504 | 0.327496 | 1.139222 | -0.139222 |
| 2000-01-10 | 0.354653 | 1.354653  | 0.563622 | 1.563622 | 0.365106 | 0.634894  |



Passing multiple functions to a Series will yield a DataFrame. The resulting column names will be the transforming functions.

```
In [185]: tsdf.A.transform([np.abs, lambda x: x+1])
```

```
Out [185]:
```

|            | absolute | <lambda>  |
|------------|----------|-----------|
| 2000-01-01 | 0.578465 | 0.421535  |
| 2000-01-02 | 0.767147 | 0.232853  |
| 2000-01-03 | 0.195348 | 1.195348  |
| 2000-01-04 | NaN      | NaN       |
| 2000-01-05 | NaN      | NaN       |
| 2000-01-06 | NaN      | NaN       |
| 2000-01-07 | NaN      | NaN       |
| 2000-01-08 | 0.556397 | 0.443603  |
| 2000-01-09 | 1.010924 | -0.010924 |
| 2000-01-10 | 0.354653 | 1.354653  |

### 9.6.4.2 Transforming with a dict

Passing a dict of functions will allow selective transforming per column.

```
In [186]: tsdf.transform({'A': np.abs, 'B': lambda x: x+1})
```

```
Out [186]:
```

|            | A        | B        |
|------------|----------|----------|
| 2000-01-01 | 0.578465 | 0.496665 |
| 2000-01-02 | 0.767147 | 0.733954 |
| 2000-01-03 | 0.195348 | 1.722247 |
| 2000-01-04 | NaN      | NaN      |
| 2000-01-05 | NaN      | NaN      |
| 2000-01-06 | NaN      | NaN      |
| 2000-01-07 | NaN      | NaN      |
| 2000-01-08 | 0.556397 | 1.542165 |
| 2000-01-09 | 1.010924 | 0.327496 |
| 2000-01-10 | 0.354653 | 1.563622 |

Passing a dict of lists will generate a multi-indexed DataFrame with these selective transforms.

```
In [187]: tsdf.transform({'A': np.abs, 'B': [lambda x: x+1, 'sqrt']})
```

```
Out [187]:
```

|            | A        | B        |          |
|------------|----------|----------|----------|
|            | absolute | <lambda> | sqrt     |
| 2000-01-01 | 0.578465 | 0.496665 | NaN      |
| 2000-01-02 | 0.767147 | 0.733954 | NaN      |
| 2000-01-03 | 0.195348 | 1.722247 | 0.849851 |
| 2000-01-04 | NaN      | NaN      | NaN      |
| 2000-01-05 | NaN      | NaN      | NaN      |
| 2000-01-06 | NaN      | NaN      | NaN      |
| 2000-01-07 | NaN      | NaN      | NaN      |
| 2000-01-08 | 0.556397 | 1.542165 | 0.736318 |
| 2000-01-09 | 1.010924 | 0.327496 | NaN      |
| 2000-01-10 | 0.354653 | 1.563622 | 0.750748 |

### 9.6.5 Applying Elementwise Functions

Since not all functions can be vectorized (accept NumPy arrays and return another array or value), the methods `applymap()` on DataFrame and analogously `map()` on Series accept any Python function taking a single value and

returning a single value. For example:

```
In [188]: df4
Out[188]:
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | -1.101558 | 1.124472  | NaN       |
| b | -0.177289 | 2.487104  | -0.634293 |
| c | 0.462215  | -0.486066 | 1.931194  |
| d | NaN       | -0.456288 | -1.222918 |

```
In [189]: f = lambda x: len(str(x))

In [190]: df4['one'].map(f)
Out[190]:
```

|   |    |
|---|----|
| a | 19 |
| b | 20 |
| c | 18 |
| d | 3  |

```
Name: one, dtype: int64

In [191]: df4.applymap(f)
Out[191]:
```

|   | one | two | three |
|---|-----|-----|-------|
| a | 19  | 18  | 3     |
| b | 20  | 18  | 19    |
| c | 18  | 20  | 18    |
| d | 3   | 19  | 19    |

`Series.map()` has an additional feature; it can be used to easily “link” or “map” values defined by a secondary series. This is closely related to *merging/joining functionality*:

```
In [192]: s = pd.Series(['six', 'seven', 'six', 'seven', 'six'],
.....:                  index=['a', 'b', 'c', 'd', 'e'])
.....:

In [193]: t = pd.Series({'six' : 6., 'seven' : 7.})

In [194]: s
Out[194]:
```

|   |       |
|---|-------|
| a | six   |
| b | seven |
| c | six   |
| d | seven |
| e | six   |

```
dtype: object

In [195]: s.map(t)
Out[195]:
```

|   |     |
|---|-----|
| a | 6.0 |
| b | 7.0 |
| c | 6.0 |
| d | 7.0 |
| e | 6.0 |

```
dtype: float64
```

```
In [196]: import pandas.util.testing as tm

In [197]: panel = tm.makePanel(5)

In [198]: panel
Out[198]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [199]: panel['ItemA']
////////////////////////////////////
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-03 | 1.092702  | 0.604244  | -2.927808 | 0.339642  |
| 2000-01-04 | -1.481449 | -0.487265 | 0.082065  | 1.499953  |
| 2000-01-05 | 1.781190  | 1.990533  | 0.456554  | -0.317818 |
| 2000-01-06 | -0.031543 | 0.327007  | -1.757911 | 0.447371  |
| 2000-01-07 | 0.480993  | 1.053639  | 0.982407  | -1.315799 |

```
In [200]: result = panel.apply(lambda x: x*2, axis='items')

In [201]: result
Out[201]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [202]: result['ItemA']
////////////////////////////////////
↪
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-03 | 2.185405  | 1.208489  | -5.855616 | 0.679285  |
| 2000-01-04 | -2.962899 | -0.974530 | 0.164130  | 2.999905  |
| 2000-01-05 | 3.562379  | 3.981066  | 0.913107  | -0.635635 |
| 2000-01-06 | -0.063086 | 0.654013  | -3.515821 | 0.894742  |
| 2000-01-07 | 0.961986  | 2.107278  | 1.964815  | -2.631598 |

```
In [203]: panel.apply(lambda x: x.dtype, axis='items')
Out[203]:
```

|            | A       | B       | C       | D       |
|------------|---------|---------|---------|---------|
| 2000-01-03 | float64 | float64 | float64 | float64 |
| 2000-01-04 | float64 | float64 | float64 | float64 |
| 2000-01-05 | float64 | float64 | float64 | float64 |

## 9.6. Function application

(continued from previous page)

|            |         |         |         |         |
|------------|---------|---------|---------|---------|
| 2000-01-06 | float64 | float64 | float64 | float64 |
| 2000-01-07 | float64 | float64 | float64 | float64 |

A similar reduction type operation.

```
In [204]: panel.apply(lambda x: x.sum(), axis='major_axis')
Out[204]:
```

|   | ItemA     | ItemB     | ItemC     |
|---|-----------|-----------|-----------|
| A | 1.841893  | 0.918017  | -1.160547 |
| B | 3.488158  | -2.629773 | 0.603397  |
| C | -3.164692 | 0.805970  | 0.806501  |
| D | 0.653349  | -0.152299 | 0.252577  |

This last reduction is equivalent to:

```
In [205]: panel.sum('major_axis')
Out[205]:
```

|   | ItemA     | ItemB     | ItemC     |
|---|-----------|-----------|-----------|
| A | 1.841893  | 0.918017  | -1.160547 |
| B | 3.488158  | -2.629773 | 0.603397  |
| C | -3.164692 | 0.805970  | 0.806501  |
| D | 0.653349  | -0.152299 | 0.252577  |

A transformation operation that returns a `Panel`, but is computing the z-score across the `major_axis`.

```
In [206]: result = panel.apply(
.....:         lambda x: (x-x.mean())/x.std(),
.....:         axis='major_axis')
.....:

In [207]: result
Out[207]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: A to D

In [208]: result['ItemA']
////////////////////////////////////
↪
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-03 | 0.585813  | -0.102070 | -1.394063 | 0.201263  |
| 2000-01-04 | -1.496089 | -1.295066 | 0.434343  | 1.318766  |
| 2000-01-05 | 1.142642  | 1.413112  | 0.661833  | -0.431942 |
| 2000-01-06 | -0.323445 | -0.405085 | -0.683386 | 0.305017  |
| 2000-01-07 | 0.091079  | 0.389108  | 0.981273  | -1.393105 |

Apply can also accept multiple axes in the `axis` argument. This will pass a `DataFrame` of the cross-section to the applied function.

```
In [209]: f = lambda x: ((x.T-x.mean(1))/x.std(1)).T
In [210]: result = panel.apply(f, axis = ['items','major_axis'])
In [211]: result
```

(continues on next page)

(continued from previous page)

```
Out [211]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC
```

```
In [212]: result.loc[:, :, 'ItemA']
```

```

////////////////////////////////////
↪
      A      B      C      D
2000-01-03  0.859304  0.448509 -1.109374  0.397237
2000-01-04 -1.053319 -1.063370  0.986639  1.152266
2000-01-05  1.106511  1.143185 -0.093917 -0.583083
2000-01-06  0.561619 -0.835608 -1.075936  0.194525
2000-01-07 -0.339514  1.097901  0.747522 -1.147605
```

This is equivalent to the following:

```
In [213]: result = pd.Panel(dict([ (ax, f(panel.loc[:, :, ax]))
.....:                             for ax in panel.minor_axis ]))
.....:
```

```
In [214]: result
```

```
Out [214]:
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 5 (major_axis) x 3 (minor_axis)
Items axis: A to D
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-07 00:00:00
Minor_axis axis: ItemA to ItemC
```

```
In [215]: result.loc[:, :, 'ItemA']
```

```

////////////////////////////////////
↪
      A      B      C      D
2000-01-03  0.859304  0.448509 -1.109374  0.397237
2000-01-04 -1.053319 -1.063370  0.986639  1.152266
2000-01-05  1.106511  1.143185 -0.093917 -0.583083
2000-01-06  0.561619 -0.835608 -1.075936  0.194525
2000-01-07 -0.339514  1.097901  0.747522 -1.147605
```

## 9.7 Reindexing and altering labels

`reindex()` is the fundamental data alignment method in pandas. It is used to implement nearly all other features relying on label-alignment functionality. To *reindex* means to conform the data to match a given set of labels along a particular axis. This accomplishes several things:

- Reorders the existing data to match a new set of labels
- Inserts missing value (NA) markers in label locations where no data for that label existed
- If specified, **fill** data for missing labels using logic (highly relevant to working with time series data)

Here is a simple example:

```
In [216]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

```
In [217]: s
```

```
Out[217]:
a    -0.454087
b    -0.360309
c    -0.951631
d    -0.535459
e     0.835231
dtype: float64
```

```
In [218]: s.reindex(['e', 'b', 'f', 'd'])
```

```
Out[218]:
e     0.835231
b    -0.360309
f         NaN
d    -0.535459
dtype: float64
```

Here, the `f` label was not contained in the Series and hence appears as `NaN` in the result.

With a `DataFrame`, you can simultaneously reindex the index and columns:

```
In [219]: df
```

```
Out[219]:
   one    two    three
a -1.101558  1.124472    NaN
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194
d         NaN -0.456288 -1.222918
```

```
In [220]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
```

```
Out[220]:
   three    two    one
c  1.931194 -0.486066  0.462215
f         NaN         NaN         NaN
b -0.634293  2.487104 -0.177289
```

You may also use `reindex` with an `axis` keyword:

```
In [221]: df.reindex(['c', 'f', 'b'], axis='index')
```

```
Out[221]:
   one    two    three
c  0.462215 -0.486066  1.931194
f         NaN         NaN         NaN
b -0.177289  2.487104 -0.634293
```

Note that the `Index` objects containing the actual axis labels can be **shared** between objects. So if we have a `Series` and a `DataFrame`, the following can be done:

```
In [222]: rs = s.reindex(df.index)
```

```
In [223]: rs
```

```
Out[223]:
a    -0.454087
b    -0.360309
```

(continues on next page)

(continued from previous page)

```
c    -0.951631
d    -0.535459
dtype: float64
```

```
In [224]: rs.index is df.index
```

```
Out[224]: True
```

This means that the reindexed Series's index is the same Python object as the DataFrame's index.

New in version 0.21.0.

`DataFrame.reindex()` also supports an “axis-style” calling convention, where you specify a single labels argument and the axis it applies to.

```
In [225]: df.reindex(['c', 'f', 'b'], axis='index')
```

```
Out[225]:
```

```
      one    two    three
c  0.462215 -0.486066  1.931194
f      NaN      NaN      NaN
b -0.177289  2.487104 -0.634293
```

```
In [226]: df.reindex(['three', 'two', 'one'], axis='columns')
```

```
Out[226]:
```

```
      three    two    one
a      NaN  1.124472 -1.101558
b -0.634293  2.487104 -0.177289
c  1.931194 -0.486066  0.462215
d -1.222918 -0.456288      NaN
```

### See also:

*MultiIndex / Advanced Indexing* is an even more concise way of doing reindexing.

**Note:** When writing performance-sensitive code, there is a good reason to spend some time becoming a reindexing ninja: **many operations are faster on pre-aligned data**. Adding two unaligned DataFrames internally triggers a reindexing step. For exploratory analysis you will hardly notice the difference (because `reindex` has been heavily optimized), but when CPU cycles matter sprinkling a few explicit `reindex` calls here and there can have an impact.

## 9.7.1 Reindexing to align with another object

You may wish to take an object and reindex its axes to be labeled the same as another object. While the syntax for this is straightforward albeit verbose, it is a common enough operation that the `reindex_like()` method is available to make this simpler:

```
In [227]: df2
```

```
Out[227]:
```

```
      one    two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066
```

```
In [228]: df3
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
      one      two
a -0.829347  0.082635
b  0.094922  1.445267
c  0.734426 -1.527903

In [229]: df.reindex_like(df2)
////////////////////////////////////
↪
      one      two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066

```

## 9.7.2 Aligning objects with each other with `align`

The `align()` method is the fastest way to simultaneously align two objects. It supports a `join` argument (related to *joining and merging*):

- `join='outer'`: take the union of the indexes (default)
- `join='left'`: use the calling object's index
- `join='right'`: use the passed object's index
- `join='inner'`: intersect the indexes

It returns a tuple with both of the reindexed Series:

```

In [230]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [231]: s1 = s[:4]

In [232]: s2 = s[1:]

In [233]: s1.align(s2)
Out[233]:
(a    0.505453
 b    1.788110
 c   -0.405908
 d   -0.801912
 e         NaN
dtype: float64, a         NaN
 b    1.788110
 c   -0.405908
 d   -0.801912
 e    0.768460
dtype: float64)

In [234]: s1.align(s2, join='inner')
////////////////////////////////////
↪
(b    1.788110
 c   -0.405908
 d   -0.801912
dtype: float64, b    1.788110

```

(continues on next page)



(continued from previous page)

```
c    -0.405908
d    -0.801912
dtype: float64)
```

```
In [235]: s1.align(s2, join='left')
```

```
////////////////////////////////////
↪
(a    0.505453
 b    1.788110
 c    -0.405908
 d    -0.801912
 dtype: float64, a          NaN
 b    1.788110
 c    -0.405908
 d    -0.801912
 dtype: float64)
```

For DataFrames, the join method will be applied to both the index and the columns by default:

```
In [236]: df.align(df2, join='inner')
```

```
Out [236]:
```

```
(      one      two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066,      one      two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066)
```

You can also pass an axis option to only align on the specified axis:

```
In [237]: df.align(df2, join='inner', axis=0)
```

```
Out [237]:
```

```
(      one      two      three
a -1.101558  1.124472          NaN
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194,      one      two
a -1.101558  1.124472
b -0.177289  2.487104
c  0.462215 -0.486066)
```

If you pass a Series to `DataFrame.align()`, you can choose to align both objects either on the DataFrame's index or columns using the axis argument:

```
In [238]: df.align(df2.iloc[0], axis=1)
```

```
Out [238]:
```

```
(      one      three      two
a -1.101558          NaN  1.124472
b -0.177289 -0.634293  2.487104
c  0.462215  1.931194 -0.486066
d          NaN -1.222918 -0.456288, one      -1.101558
three          NaN
two          1.124472
Name: a, dtype: float64)
```

### 9.7.3 Filling while reindexing

`reindex()` takes an optional parameter `method` which is a filling method chosen from the following table:

| Method           | Action                            |
|------------------|-----------------------------------|
| pad / ffill      | Fill values forward               |
| bfill / backfill | Fill values backward              |
| nearest          | Fill from the nearest index value |

We illustrate these fill methods on a simple Series:

```
In [239]: rng = pd.date_range('1/3/2000', periods=8)
```

```
In [240]: ts = pd.Series(np.random.randn(8), index=rng)
```

```
In [241]: ts2 = ts[[0, 3, 6]]
```

```
In [242]: ts
```

```
Out [242]:
2000-01-03    0.466284
2000-01-04   -0.457411
2000-01-05   -0.364060
2000-01-06    0.785367
2000-01-07   -1.463093
2000-01-08    1.187315
2000-01-09   -0.493153
2000-01-10   -1.323445
Freq: D, dtype: float64
```

```
In [243]: ts2
```

```

////////////////////////////////////
↪
2000-01-03    0.466284
2000-01-06    0.785367
2000-01-09   -0.493153
dtype: float64
```

```
In [244]: ts2.reindex(ts.index)
```

```

////////////////////////////////////
↪
2000-01-03    0.466284
2000-01-04         NaN
2000-01-05         NaN
2000-01-06    0.785367
2000-01-07         NaN
2000-01-08         NaN
2000-01-09   -0.493153
2000-01-10         NaN
Freq: D, dtype: float64
```

```
In [245]: ts2.reindex(ts.index, method='ffill')
```

```

////////////////////////////////////
↪
2000-01-03    0.466284
2000-01-04    0.466284
2000-01-05    0.466284
2000-01-06    0.785367
```

(continues on next page)

(continued from previous page)

```

2000-01-07    0.785367
2000-01-08    0.785367
2000-01-09   -0.493153
2000-01-10   -0.493153
Freq: D, dtype: float64

```

```
In [246]: ts2.reindex(ts.index, method='bfill')
```

```

////////////////////////////////////
↪
2000-01-03    0.466284
2000-01-04    0.785367
2000-01-05    0.785367
2000-01-06    0.785367
2000-01-07   -0.493153
2000-01-08   -0.493153
2000-01-09   -0.493153
2000-01-10         NaN
Freq: D, dtype: float64

```

```
In [247]: ts2.reindex(ts.index, method='nearest')
```

```

////////////////////////////////////
↪
2000-01-03    0.466284
2000-01-04    0.466284
2000-01-05    0.785367
2000-01-06    0.785367
2000-01-07    0.785367
2000-01-08   -0.493153
2000-01-09   -0.493153
2000-01-10   -0.493153
Freq: D, dtype: float64

```

These methods require that the indexes are **ordered** increasing or decreasing.

Note that the same result could have been achieved using *fillna* (except for `method='nearest'`) or *interpolate*:

```
In [248]: ts2.reindex(ts.index).fillna(method='ffill')
```

```
Out [248]:
```

```

2000-01-03    0.466284
2000-01-04    0.466284
2000-01-05    0.466284
2000-01-06    0.785367
2000-01-07    0.785367
2000-01-08    0.785367
2000-01-09   -0.493153
2000-01-10   -0.493153
Freq: D, dtype: float64

```

*reindex()* will raise a `ValueError` if the index is not monotonically increasing or decreasing. *fillna()* and *interpolate()* will not perform any checks on the order of the index.

## 9.7.4 Limits on filling while reindexing

The `limit` and `tolerance` arguments provide additional control over filling while reindexing. `Limit` specifies the maximum count of consecutive matches:

```
In [249]: ts2.reindex(ts.index, method='ffill', limit=1)
Out[249]:
2000-01-03    0.466284
2000-01-04    0.466284
2000-01-05         NaN
2000-01-06    0.785367
2000-01-07    0.785367
2000-01-08         NaN
2000-01-09   -0.493153
2000-01-10   -0.493153
Freq: D, dtype: float64
```

In contrast, `tolerance` specifies the maximum distance between the index and indexer values:

```
In [250]: ts2.reindex(ts.index, method='ffill', tolerance='1 day')
Out[250]:
2000-01-03    0.466284
2000-01-04    0.466284
2000-01-05         NaN
2000-01-06    0.785367
2000-01-07    0.785367
2000-01-08         NaN
2000-01-09   -0.493153
2000-01-10   -0.493153
Freq: D, dtype: float64
```

Notice that when used on a `DatetimeIndex`, `TimedeltaIndex` or `PeriodIndex`, `tolerance` will be coerced into a `Timedelta` if possible. This allows you to specify tolerance with appropriate strings.

## 9.7.5 Dropping labels from an axis

A method closely related to `reindex` is the `drop()` function. It removes a set of labels from an axis:

```
In [251]: df
Out[251]:
      one      two      three
a -1.101558  1.124472      NaN
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194
d         NaN -0.456288 -1.222918

In [252]: df.drop(['a', 'd'], axis=0)
↪
      one      two      three
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194

In [253]: df.drop(['one'], axis=1)
↪
      two      three
a  1.124472      NaN
b  2.487104 -0.634293
c -0.486066  1.931194
d -0.456288 -1.222918
```

Note that the following also works, but is a bit less obvious / clean:

```
In [254]: df.reindex(df.index.difference(['a', 'd']))
Out[254]:
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| b | -0.177289 | 2.487104  | -0.634293 |
| c | 0.462215  | -0.486066 | 1.931194  |

## 9.7.6 Renaming / mapping labels

The `rename()` method allows you to relabel an axis based on some mapping (a dict or Series) or an arbitrary function.

```
In [255]: s
Out[255]:
```

| a | 0.505453  |
|---|-----------|
| b | 1.788110  |
| c | -0.405908 |
| d | -0.801912 |
| e | 0.768460  |

dtype: float64

```
In [256]: s.rename(str.upper)
Out[256]:
```

| A | 0.505453  |
|---|-----------|
| B | 1.788110  |
| C | -0.405908 |
| D | -0.801912 |
| E | 0.768460  |

dtype: float64

If you pass a function, it must return a value when called with any of the labels (and must produce a set of unique values). A dict or Series can also be used:

```
In [257]: df.rename(columns={'one': 'foo', 'two': 'bar'},
.....:               index={'a': 'apple', 'b': 'banana', 'd': 'durian'})
Out[257]:
```

|        | foo       | bar       | three     |
|--------|-----------|-----------|-----------|
| apple  | -1.101558 | 1.124472  | NaN       |
| banana | -0.177289 | 2.487104  | -0.634293 |
| c      | 0.462215  | -0.486066 | 1.931194  |
| durian | NaN       | -0.456288 | -1.222918 |

If the mapping doesn't include a column/index label, it isn't renamed. Note that extra labels in the mapping don't throw an error.

New in version 0.21.0.

`DataFrame.rename()` also supports an “axis-style” calling convention, where you specify a single mapper and the axis to apply that mapping to.

```
In [258]: df.rename({'one': 'foo', 'two': 'bar'}, axis='columns')
Out[258]:
```

|   | foo       | bar      | three     |
|---|-----------|----------|-----------|
| a | -1.101558 | 1.124472 | NaN       |
| b | -0.177289 | 2.487104 | -0.634293 |

(continues on next page)

(continued from previous page)

```
c  0.462215 -0.486066  1.931194
d      NaN -0.456288 -1.222918
```

```
In [259]: df.rename({'a': 'apple', 'b': 'banana', 'd': 'durian'}, axis='index')
```

```

////////////////////////////////////
↪
           one      two      three
apple -1.101558  1.124472      NaN
banana -0.177289  2.487104 -0.634293
c      0.462215 -0.486066  1.931194
durian      NaN -0.456288 -1.222918
```

The `rename()` method also provides an `inplace` named parameter that is by default `False` and copies the underlying data. Pass `inplace=True` to rename the data in place.

New in version 0.18.0.

Finally, `rename()` also accepts a scalar or list-like for altering the `Series.name` attribute.

```
In [260]: s.rename("scalar-name")
```

```
Out[260]:
```

```
a      0.505453
b      1.788110
c     -0.405908
d     -0.801912
e      0.768460
Name: scalar-name, dtype: float64
```

The `Panel` class has a related `rename_axis()` class which can rename any of its three axes.

## 9.8 Iteration

The behavior of basic iteration over pandas objects depends on the type. When iterating over a `Series`, it is regarded as array-like, and basic iteration produces the values. Other data structures, like `DataFrame` and `Panel`, follow the dict-like convention of iterating over the “keys” of the objects.

In short, basic iteration (`for i in object`) produces:

- **Series:** values
- **DataFrame:** column labels
- **Panel:** item labels

Thus, for example, iterating over a `DataFrame` gives you the column names:

```
In [261]: df = pd.DataFrame({'col1' : np.random.randn(3), 'col2' : np.random.randn(3)})
↪,
.....:                index=['a', 'b', 'c'])
.....:
```

```
In [262]: for col in df:
.....:     print(col)
.....:
```

```
col1
col2
```

Pandas objects also have the dict-like `iteritems()` method to iterate over the (key, value) pairs.

To iterate over the rows of a DataFrame, you can use the following methods:

- `iterrows()`: Iterate over the rows of a DataFrame as (index, Series) pairs. This converts the rows to Series objects, which can change the dtypes and has some performance implications.
- `itertuples()`: Iterate over the rows of a DataFrame as namedtuples of the values. This is a lot faster than `iterrows()`, and is in most cases preferable to use to iterate over the values of a DataFrame.

**Warning:** Iterating through pandas objects is generally **slow**. In many cases, iterating manually over the rows is not needed and can be avoided with one of the following approaches:

- Look for a *vectorized* solution: many operations can be performed using built-in methods or NumPy functions, (boolean) indexing, ...
- When you have a function that cannot work on the full DataFrame/Series at once, it is better to use `apply()` instead of iterating over the values. See the docs on *function application*.
- If you need to do iterative manipulations on the values but performance is important, consider writing the inner loop with cython or numba. See the *enhancing performance* section for some examples of this approach.

**Warning:** You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect!

For example, in the following case setting the value has no effect:

```
In [263]: df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})

In [264]: for index, row in df.iterrows():
.....:     row['a'] = 10
.....:

In [265]: df
Out[265]:
   a  b
0  1  a
1  2  b
2  3  c
```

### 9.8.1 iteritems

Consistent with the dict-like interface, `iteritems()` iterates through key-value pairs:

- **Series:** (index, scalar value) pairs
- **DataFrame:** (column, Series) pairs
- **Panel:** (item, DataFrame) pairs

For example:

```
In [266]: for item, frame in wp.iteritems():
.....:     print(item)
.....:     print(frame)
.....:
```

(continues on next page)

(continued from previous page)

```

Item1
      A      B      C      D
2000-01-01 -0.433567 -0.273610  0.680433 -0.308450
2000-01-02 -0.276099 -1.821168 -1.993606 -1.927385
2000-01-03 -2.027924  1.624972  0.551135  3.059267
2000-01-04  0.455264 -0.030740  0.935716  1.061192
2000-01-05 -2.107852  0.199905  0.323586 -0.641630
Item2
      A      B      C      D
2000-01-01 -0.587514  0.053897  0.194889 -0.381994
2000-01-02  0.318587  2.089075 -0.728293 -0.090255
2000-01-03 -0.748199  1.318931 -2.029766  0.792652
2000-01-04  0.461007 -0.542749 -0.305384 -0.479195
2000-01-05  0.095031 -0.270099 -0.707140 -0.773882

```

## 9.8.2 iterrows

`iterrows()` allows you to iterate through the rows of a DataFrame as Series objects. It returns an iterator yielding each index value along with a Series containing the data in each row:

```

In [267]: for row_index, row in df.iterrows():
.....:     print('%s\n%s' % (row_index, row))
.....:
0
a    1
b    a
Name: 0, dtype: object
1
a    2
b    b
Name: 1, dtype: object
2
a    3
b    c
Name: 2, dtype: object

```

**Note:** Because `iterrows()` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```

In [268]: df_orig = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])

In [269]: df_orig.dtypes
Out[269]:
int      int64
float    float64
dtype: object

In [270]: row = next(df_orig.iterrows())[1]

In [271]: row
Out[271]:
int      1.0
float    1.5
Name: 0, dtype: float64

```



All values in `row`, returned as a Series, are now upcasted to floats, also the original integer value in column `x`:

```
In [272]: row['int'].dtype
Out[272]: dtype('float64')

In [273]: df_orig['int'].dtype
Out[273]: dtype('int64')
```

To preserve dtypes while iterating over the rows, it is better to use `itertuples()` which returns namedtuples of the values and which is generally much faster than `iterrows()`.

For instance, a contrived way to transpose the DataFrame would be:

```
In [274]: df2 = pd.DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})  
  
In [275]: print(df2)  
      x  y  
0     1  4  
1     2  5  
2     3  6  
  
In [276]: print(df2.T)  
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\    0   1   2  
x     1   2   3  
y     4   5   6  
  
In [277]: df2_t = pd.DataFrame(dict((idx, values) for idx, values in df2.iterrows()))  
  
In [278]: print(df2_t)  
      0   1   2  
x     1   2   3  
y     4   5   6
```

### 9.8.3 itertuples

The `itertuples()` method will return an iterator yielding a namedtuple for each row in the DataFrame. The first element of the tuple will be the row's corresponding index value, while the remaining values are the row values.

For instance:

```
In [279]: for row in df.itertuples():
.....:     print(row)
.....:
Pandas(Index=0, a=1, b='a')
Pandas(Index=1, a=2, b='b')
Pandas(Index=2, a=3, b='c')
```

This method does not convert the row to a Series object; it merely returns the values inside a namedtuple. Therefore, `itertuples()` preserves the data type of the values and is generally faster as `iterrows()`.

**Note:** The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

## 9.9 .dt accessor

Series has an accessor to succinctly return datetime like properties for the *values* of the Series, if it is a date-time/period like Series. This will return a Series, indexed like the existing Series.

```
# datetime
In [280]: s = pd.Series(pd.date_range('20130101 09:10:12', periods=4))

In [281]: s
Out[281]:
0    2013-01-01 09:10:12
1    2013-01-02 09:10:12
2    2013-01-03 09:10:12
3    2013-01-04 09:10:12
dtype: datetime64[ns]

In [282]: s.dt.hour
Out[282]:
0    9
1    9
2    9
3    9
dtype: int64

In [283]: s.dt.second
Out[283]:
0    12
1    12
2    12
3    12
dtype: int64

In [284]: s.dt.day
Out[284]:
0    1
1    2
2    3
3    4
dtype: int64
```

This enables nice expressions like this:

```
In [285]: s[s.dt.day==2]
Out[285]:
1    2013-01-02 09:10:12
dtype: datetime64[ns]
```

You can easily produces tz aware transformations:

```
In [286]: stz = s.dt.tz_localize('US/Eastern')

In [287]: stz
Out[287]:
0    2013-01-01 09:10:12-05:00
```

(continues on next page)

(continued from previous page)

```

1    2013-01-02 09:10:12-05:00
2    2013-01-03 09:10:12-05:00
3    2013-01-04 09:10:12-05:00
dtype: datetime64[ns, US/Eastern]

```

```
In [288]: stz.dt.tz
```

```

////////////////////////////////////
↪ <DstTzInfo 'US/Eastern' LMT-1 day, 19:04:00 STD>

```

You can also chain these types of operations:

```
In [289]: s.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
```

```
Out[289]:
```

```

0    2013-01-01 04:10:12-05:00
1    2013-01-02 04:10:12-05:00
2    2013-01-03 04:10:12-05:00
3    2013-01-04 04:10:12-05:00
dtype: datetime64[ns, US/Eastern]

```

You can also format datetime values as strings with `Series.dt.strftime()` which supports the same format as the standard `strftime()`.

```
# DatetimeIndex
```

```
In [290]: s = pd.Series(pd.date_range('20130101', periods=4))
```

```
In [291]: s
```

```
Out[291]:
```

```

0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: datetime64[ns]

```

```
In [292]: s.dt.strftime('%Y/%m/%d')
```

```

////////////////////////////////////Out [292]:
↪
0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object

```

```
# PeriodIndex
```

```
In [293]: s = pd.Series(pd.period_range('20130101', periods=4))
```

```
In [294]: s
```

```
Out[294]:
```

```

0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: object

```

```
In [295]: s.dt.strftime('%Y/%m/%d')
```

```

////////////////////////////////////Out [295]:
↪

```

(continues on next page)

(continued from previous page)

```

0    2013/01/01
1    2013/01/02
2    2013/01/03
3    2013/01/04
dtype: object

```

The `.dt` accessor works for period and timedelta dtypes.

```

# period
In [296]: s = pd.Series(pd.period_range('20130101', periods=4, freq='D'))

In [297]: s
Out[297]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
3    2013-01-04
dtype: object

In [298]: s.dt.year
Out[298]:
0    2013
1    2013
2    2013
3    2013
dtype: int64

In [299]: s.dt.day
Out[299]:
0    1
1    2
2    3
3    4
dtype: int64

```

```

# timedelta
In [300]: s = pd.Series(pd.timedelta_range('1 day 00:00:05', periods=4, freq='s'))

In [301]: s
Out[301]:
0    1 days 00:00:05
1    1 days 00:00:06
2    1 days 00:00:07
3    1 days 00:00:08
dtype: timedelta64[ns]

In [302]: s.dt.days
Out[302]:
0    1
1    1
2    1
3    1
dtype: int64

```

(continues on next page)

(continued from previous page)

**In [303]:** s.dt.seconds

```

////////////////////////////////////
↪
0      5
1      6
2      7
3      8
dtype: int64

```

**In [304]:** s.dt.components

```

////////////////////////////////////
↪
   days  hours  minutes  seconds  milliseconds  microseconds  nanoseconds
0     1     0         0         5             0             0             0
1     1     0         0         6             0             0             0
2     1     0         0         7             0             0             0
3     1     0         0         8             0             0             0

```

---

**Note:** `Series.dt` will raise a `TypeError` if you access with a non-datetime-like values.

---

## 9.10 Vectorized string methods

Series is equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the Series's `str` attribute and generally have names matching the equivalent (scalar) built-in string methods. For example:

```

In [305]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog
↪', 'cat'])

```

**In [306]:** s.str.lower()**Out [306]:**

```

0      a
1      b
2      c
3    aaba
4    baca
5     NaN
6    caba
7    dog
8    cat
dtype: object

```

Powerful pattern-matching methods are provided as well, but note that pattern-matching generally uses [regular expressions](#) by default (and in some cases always uses them).

Please see [Vectorized String Methods](#) for a complete description.

## 9.11 Sorting

Pandas supports three kinds of sorting: sorting by index labels, sorting by column values, and sorting by a combination of both.

### 9.11.1 By Index

The `Series.sort_index()` and `DataFrame.sort_index()` methods are used to sort a pandas object by its index levels.

```
In [307]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3), index=['a', 'b', 'c',
↳ '']),
↳ .....:                                'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c',
↳ 'd']),
↳ .....:                                'three' : pd.Series(np.random.randn(3), index=['b', 'c',
↳ 'd'])})
↳ .....:
```

[illegible]

```
In [309]: unsorted_df
Out[309]:
```

|   | three     | two       | one       |
|---|-----------|-----------|-----------|
| a | NaN       | 0.708543  | 0.036274  |
| d | -0.540166 | 0.586626  | NaN       |
| c | 0.410238  | 1.121731  | 1.044630  |
| b | -0.282532 | -2.038777 | -0.490032 |

```
# DataFrame
```

```
In [310]: unsorted_df.sort_index()
```

```
In [311]: unsorted_df.sort_index(ascending=False)
```

```
In [312]: unsorted_df.sort_index(axis=1)
```

(continues on next page)

(continued from previous page)

```
b -0.490032 -0.282532 -2.038777
```

```
# Series
```

```
In [313]: unsorted_df['three'].sort_index()
```

```
////////////////////////////////////
```

```
↪
```

```
a      NaN
```

```
b    -0.282532
```

```
c     0.410238
```

```
d    -0.540166
```

```
Name: three, dtype: float64
```

### 9.11.2 By Values

The `Series.sort_values()` method is used to sort a *Series* by its values. The `DataFrame.sort_values()` method is used to sort a *DataFrame* by its column or row values. The optional `by` parameter to `DataFrame.sort_values()` may be used to specify one or more columns to use to determine the sorted order.

```
In [314]: df1 = pd.DataFrame({'one': [2, 1, 1, 1], 'two': [1, 3, 2, 4], 'three': [5, 4, 3, 2]})
```

```
In [315]: df1.sort_values(by='two')
```

```
Out[315]:
```

|   | one | two | three |
|---|-----|-----|-------|
| 0 | 2   | 1   | 5     |
| 2 | 1   | 2   | 3     |
| 1 | 1   | 3   | 4     |
| 3 | 1   | 4   | 2     |

The `by` parameter can take a list of column names, e.g.:

```
In [316]: df1[['one', 'two', 'three']].sort_values(by=['one', 'two'])
```

```
Out[316]:
```

|   | one | two | three |
|---|-----|-----|-------|
| 2 | 1   | 2   | 3     |
| 1 | 1   | 3   | 4     |
| 3 | 1   | 4   | 2     |
| 0 | 2   | 1   | 5     |

These methods have special treatment of NA values via the `na_position` argument:

```
In [317]: s[2] = np.nan
```

```
In [318]: s.sort_values()
```

```
Out[318]:
```

|   |      |
|---|------|
| 0 | A    |
| 3 | Aaba |
| 1 | B    |
| 4 | Baca |
| 6 | CABA |
| 8 | cat  |
| 7 | dog  |
| 2 | NaN  |
| 5 | NaN  |

dtype: object

(continues on next page)

(continued from previous page)

```
In [319]: s.sort_values(na_position='first')
```

```

////////////////////////////////////
↪
2      NaN
5      NaN
0       A
3     Aaba
1       B
4     Baca
6     CABA
8     cat
7     dog
dtype: object

```

### 9.11.3 By Indexes and Values

New in version 0.23.0.

Strings passed as the `by` parameter to `DataFrame.sort_values()` may refer to either columns or index level names.

```

# Build MultiIndex
In [320]: idx = pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('a', 2),
.....:                                   ('b', 2), ('b', 1), ('b', 1)])
.....:

In [321]: idx.names = ['first', 'second']

# Build DataFrame
In [322]: df_multi = pd.DataFrame({'A': np.arange(6, 0, -1)},
.....:                             index=idx)
.....:

In [323]: df_multi
Out[323]:

```

|   | first | second | A |
|---|-------|--------|---|
| a | 1     | 2      | 6 |
|   | 2     | 2      | 5 |
|   | 2     | 4      | 4 |
| b | 2     | 3      | 3 |
|   | 1     | 2      | 2 |
|   | 1     | 1      | 1 |

Sort by 'second' (index) and 'A' (column)

```
In [324]: df_multi.sort_values(by=['second', 'A'])
Out[324]:
```

|   | first | second | A |
|---|-------|--------|---|
| b | 1     | 1      | 1 |
|   | 1     | 2      | 2 |
| a | 1     | 2      | 6 |
| b | 2     | 3      | 3 |
| a | 2     | 4      | 4 |
|   | 2     | 5      | 5 |



Series has the `searchsorted()` method, which works similarly to `numpy.ndarray.searchsorted()`.

### 9.11.5 smallest / largest values

```
In [332]: s = pd.Series(np.random.permutation(10))

In [333]: s
Out[333]:
0      8
1      2
2      9
3      5
4      6
5      0
6      1
7      7
8      4
9      3
dtype: int64

In [334]: s.sort_values()
Out[334]:
5      0
6      1
```

(continues on next page)

(continued from previous page)

```

1      2
9      3
8      4
3      5
4      6
7      7
0      8
2      9
dtype: int64

```

```
In [335]: s.nsmallest(3)
```

```

////////////////////////////////////
↪
5      0
6      1
1      2
dtype: int64

```

```
In [336]: s.nlargest(3)
```

```

////////////////////////////////////
↪
2      9
0      8
7      7
dtype: int64

```

DataFrame also has the `nlargest` and `nsmallest` methods.

```
In [337]: df = pd.DataFrame({'a': [-2, -1, 1, 10, 8, 11, -1],
.....:                      'b': list('abdceff'),
.....:                      'c': [1.0, 2.0, 4.0, 3.2, np.nan, 3.0, 4.0]})
.....:
```

```
In [338]: df.nlargest(3, 'a')
```

```
Out[338]:
   a  b    c
5  11 f  3.0
3  10 c  3.2
4   8 e  NaN

```

```
In [339]: df.nlargest(5, ['a', 'c'])
```

```

////////////////////////////////////Out[339]:
   a  b    c
6  -1 f  4.0
5  11 f  3.0
3  10 c  3.2
4   8 e  NaN
2   1 d  4.0

```

```
In [340]: df.nsmallest(3, 'a')
```

```

////////////////////////////////////
↪
   a  b    c
0  -2 a  1.0
1  -1 b  2.0
6  -1 f  4.0

```

(continues on next page)

(continued from previous page)

```
In [341]: df.nsmallest(5, ['a', 'c'])
```

```

////////////////////////////////////
↪
   a  b    c
0 -2  a  1.0
2  1  d  4.0
4  8  e  NaN
1 -1  b  2.0
6 -1  f  4.0

```

### 9.11.6 Sorting by a multi-index column

You must be explicit about sorting when the column is a multi-index, and fully specify all levels to `by`.

```
In [342]: df1.columns = pd.MultiIndex.from_tuples([('a', 'one'), ('a', 'two'), ('b', 'three')])
```

```
In [343]: df1.sort_values(by=('a', 'two'))
```

```
Out[343]:
      a      b
  one two three
0    2    1     5
2    1    2     3
1    1    3     4
3    1    4     2

```

## 9.12 Copying

The `copy()` method on pandas objects copies the underlying data (though not the axis indexes, since they are immutable) and returns a new object. Note that **it is seldom necessary to copy objects**. For example, there are only a handful of ways to alter a DataFrame *in-place*:

- Inserting, deleting, or modifying a column.
- Assigning to the `index` or `columns` attributes.
- For homogeneous data, directly modifying the values via the `values` attribute or advanced indexing.

To be clear, no pandas method has the side effect of modifying your data; almost every method returns a new object, leaving the original object untouched. If the data is modified, it is because you did so explicitly.

## 9.13 dtypes

The main types stored in pandas objects are `float`, `int`, `bool`, `datetime64[ns]` and `datetime64[ns, tz]`, `timedelta[ns]`, `category` and `object`. In addition these dtypes have item sizes, e.g. `int64` and `int32`. See [Series with TZ](#) for more detail on `datetime64[ns, tz]` dtypes.

A convenient `dtypes` attribute for DataFrame returns a Series with the data type of each column.

```
In [344]: dft = pd.DataFrame(dict(A = np.random.rand(3),
.....:                             B = 1,
```

(continues on next page)

(continued from previous page)

```

.....:
.....:
.....:
.....:
.....:
.....:
.....:
C = 'foo',
D = pd.Timestamp('20010102'),
E = pd.Series([1.0]*3).astype('float32'),
F = False,
G = pd.Series([1]*3, dtype='int8'))

In [345]: dft
Out[345]:

```

|   | A        | B | C   | D          | E   | F     | G |
|---|----------|---|-----|------------|-----|-------|---|
| 0 | 0.809585 | 1 | foo | 2001-01-02 | 1.0 | False | 1 |
| 1 | 0.128238 | 1 | foo | 2001-01-02 | 1.0 | False | 1 |
| 2 | 0.775752 | 1 | foo | 2001-01-02 | 1.0 | False | 1 |

```

In [346]: dft.dtypes
dtype: object
A          float64
B           int64
C           object
D    datetime64[ns]
E          float32
F             bool
G           int8
dtype: object

```

On a Series object, use the `dtype` attribute.

```

In [347]: dft['A'].dtype
Out[347]: dtype('float64')

```

If a pandas object contains data with multiple dtypes *in a single column*, the dtype of the column will be chosen to accommodate all of the data types (object is the most general).

```

# these ints are coerced to floats
In [348]: pd.Series([1, 2, 3, 4, 5, 6.])
Out[348]:
0    1.0
1    2.0
2    3.0
3    4.0
4    5.0
5    6.0
dtype: float64

# string data forces an ``object`` dtype
In [349]: pd.Series([1, 2, 3, 6., 'foo'])
Out[349]:
0    1
1    2
2    3
3    6
4   foo
dtype: object

```

The number of columns of each type in a DataFrame can be found by calling `get_dtype_counts()`.

```
In [350]: dft.get_dtype_counts()
Out[350]:
float64      1
float32      1
int64        1
int8         1
datetime64[ns] 1
bool         1
object       1
dtype: int64
```

Numeric dtypes will propagate and can coexist in DataFrames. If a dtype is passed (either directly via the `dtype` keyword, a passed `ndarray`, or a passed `Series`, then it will be preserved in DataFrame operations. Furthermore, different numeric dtypes will **NOT** be combined. The following example will give you a taste.

```
In [351]: df1 = pd.DataFrame(np.random.randn(8, 1), columns=['A'], dtype='float32')

In [352]: df1
Out[352]:
      A
0  0.890400
1  0.283331
2 -0.303613
3 -1.192210
4  0.065420
5  0.455918
6  2.008328
7  0.188942

In [353]: df1.dtypes
=====
A      float32
dtype: object

In [354]: df2 = pd.DataFrame(dict( A = pd.Series(np.random.randn(8), dtype='float16'),
.....:                               B = pd.Series(np.random.randn(8)),
.....:                               C = pd.Series(np.array(np.random.randn(8), dtype=
→ 'uint8')) ))
.....:

In [355]: df2
Out[355]:
      A      B      C
0 -0.454346  0.200071  255
1 -0.916504 -0.557756  255
2  0.640625 -0.141988    0
3  2.675781 -0.174060    0
4 -0.007866  0.258626    0
5 -0.204224  0.941688    0
6 -0.100098 -1.849045    0
7 -0.402100 -0.949458    0

In [356]: df2.dtypes
=====
A      float16
```

(continues on next page)

(continued from previous page)

```
B    float64
C      uint8
dtype: object
```

### 9.13.1 defaults

By default integer types are `int64` and float types are `float64`, *regardless* of platform (32-bit or 64-bit). The following will all result in `int64` dtypes.

```
In [357]: pd.DataFrame([1, 2], columns=['a']).dtypes
Out[357]:
a    int64
dtype: object

In [358]: pd.DataFrame({'a': [1, 2]}).dtypes
Out[358]:
a    int64
dtype: object

In [359]: pd.DataFrame({'a': 1 }, index=list(range(2))).dtypes
Out[359]:
a    int64
dtype: object
```

Note that Numpy will choose *platform-dependent* types when creating arrays. The following **WILL** result in `int32` on 32-bit platform.

```
In [360]: frame = pd.DataFrame(np.array([1, 2]))
```

### 9.13.2 upcasting

Types can potentially be *upcasted* when combined with other types, meaning they are promoted from the current type (e.g. `int` to `float`).

```
In [361]: df3 = df1.reindex_like(df2).fillna(value=0.0) + df2

In [362]: df3
Out[362]:
```

|   | A         | B         | C     |
|---|-----------|-----------|-------|
| 0 | 0.436054  | 0.200071  | 255.0 |
| 1 | -0.633173 | -0.557756 | 255.0 |
| 2 | 0.337012  | -0.141988 | 0.0   |
| 3 | 1.483571  | -0.174060 | 0.0   |
| 4 | 0.057555  | 0.258626  | 0.0   |
| 5 | 0.251695  | 0.941688  | 0.0   |
| 6 | 1.908231  | -1.849045 | 0.0   |
| 7 | -0.213158 | -0.949458 | 0.0   |

```
In [363]: df3.dtypes
Out[363]:
A    float32
B    float64
```

(continues on next page)

```
C      float64
dtype: object
```

```
In [364]: df3.values.dtype
Out[364]: dtype('float64')
```

```
In [365]: df3
Out[365]:
```

|   | A         | B         | C     |
|---|-----------|-----------|-------|
| 0 | 0.436054  | 0.200071  | 255.0 |
| 1 | -0.633173 | -0.557756 | 255.0 |
| 2 | 0.337012  | -0.141988 | 0.0   |
| 3 | 1.483571  | -0.174060 | 0.0   |
| 4 | 0.057555  | 0.258626  | 0.0   |
| 5 | 0.251695  | 0.941688  | 0.0   |
| 6 | 1.908231  | -1.849045 | 0.0   |
| 7 | -0.213158 | -0.949458 | 0.0   |

```

A      float32
B      float64
C      float64
dtype: object

```

```
In [367]: df3.astype('float32').dtypes
\\/////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↪
A      float32
B      float32
C      float32
dtype: object
```

(continued from previous page)

```

Out [370]:
   a  b  c
0  1  4  7
1  2  5  8
2  3  6  9

In [371]: dft.dtypes
Out [371]:
a      uint8
b      uint8
c      int64
dtype: object

```

New in version 0.19.0.

Convert certain columns to a specific dtype by passing a dict to `astype()`.

```

In [372]: dft1 = pd.DataFrame({'a': [1,0,1], 'b': [4,5,6], 'c': [7, 8, 9]})

In [373]: dft1 = dft1.astype({'a': np.bool, 'c': np.float64})

In [374]: dft1
Out [374]:
   a  b  c
0  True  4  7.0
1 False  5  8.0
2  True  6  9.0

In [375]: dft1.dtypes
Out [375]:
a      bool
b      int64
c     float64
dtype: object

```

**Note:** When trying to convert a subset of columns to a specified type using `astype()` and `loc()`, upcasting occurs. `loc()` tries to fit in what we are assigning to the current dtypes, while `[]` will overwrite them taking the dtype from the right hand side. Therefore the following piece of code produces the unintended result.

```

In [376]: dft = pd.DataFrame({'a': [1,2,3], 'b': [4,5,6], 'c': [7, 8, 9]})

In [377]: dft.loc[:, ['a', 'b']].astype(np.uint8).dtypes
Out [377]:
a      uint8
b      uint8
dtype: object

In [378]: dft.loc[:, ['a', 'b']] = dft.loc[:, ['a', 'b']].astype(np.uint8)

In [379]: dft.dtypes
Out [379]:
a      int64
b      int64
c      int64

```

(continues on next page)



(continued from previous page)

```
dtype: object
```

### 9.13.4 object conversion

pandas offers various functions to try to force conversion of types from the `object` dtype to other types. In cases where the data is already of the correct type, but stored in an `object` array, the `DataFrame.infer_objects()` and `Series.infer_objects()` methods can be used to soft convert to the correct type.

[illegible]

Because the data was transposed the original inference stored all columns as object, which `infer_objects` will correct.

```
In [385]: df.infer_objects().dtypes
Out[385]:
0          int64
1         object
2    datetime64[ns]
dtype: object
```

The following functions are available for one dimensional object arrays or scalars to perform hard conversion of objects to a specified type:

- `to_numeric()` (conversion to numeric dtypes)

```
In [386]: m = ['1.1', 2, 3]

In [387]: pd.to_numeric(m)
Out[387]: array([ 1.1,  2. ,  3. ])
```

- `to_datetime()` (conversion to datetime objects)

```
In [388]: import datetime

In [389]: m = ['2016-07-09', datetime.datetime(2016, 3, 2)]

In [390]: pd.to_datetime(m)
Out[390]: DatetimeIndex(['2016-07-09', '2016-03-02'], dtype='datetime64[ns]',
↳ freq=None)
```

- `to_timedelta()` (conversion to timedelta objects)

```
In [391]: m = ['5us', pd.Timedelta('1day')]

In [392]: pd.to_timedelta(m)
Out[392]: TimedeltaIndex(['0 days 00:00:00.000005', '1 days 00:00:00'], dtype=
↳ 'timedelta64[ns]', freq=None)
```

To force a conversion, we can pass in an `errors` argument, which specifies how pandas should deal with elements that cannot be converted to desired dtype or object. By default, `errors='raise'`, meaning that any errors encountered will be raised during the conversion process. However, if `errors='coerce'`, these errors will be ignored and pandas will convert problematic elements to `pd.NaT` (for datetime and timedelta) or `np.nan` (for numeric). This might be useful if you are reading in data which is mostly of the desired dtype (e.g. numeric, datetime), but occasionally has non-conforming elements intermixed that you want to represent as missing:

```
In [393]: import datetime

In [394]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [395]: pd.to_datetime(m, errors='coerce')
Out[395]: DatetimeIndex(['NaT', '2016-03-02'], dtype='datetime64[ns]', freq=None)

In [396]: m = ['apple', 2, 3]

In [397]: pd.to_numeric(m, errors='coerce')
Out[397]: array([ nan,   2.,   3.])

In [398]: m = ['apple', pd.Timedelta('1day')]

In [399]: pd.to_timedelta(m, errors='coerce')
Out[399]: TimedeltaIndex([NaT, '1 days'], dtype='timedelta64[ns]', freq=None)
```

The `errors` parameter has a third option of `errors='ignore'`, which will simply return the passed in data if it encounters any errors with the conversion to a desired data type:

```
In [400]: import datetime

In [401]: m = ['apple', datetime.datetime(2016, 3, 2)]

In [402]: pd.to_datetime(m, errors='ignore')
Out[402]: array(['apple', datetime.datetime(2016, 3, 2, 0, 0)], dtype=object)

In [403]: m = ['apple', 2, 3]

In [404]: pd.to_numeric(m, errors='ignore')
Out[404]: array(['apple', 2, 3], dtype=object)

In [405]: m = ['apple', pd.Timedelta('1day')]
```

(continues on next page)

(continued from previous page)

```
In [406]: pd.to_timedelta(m, errors='ignore')
Out [406]: array(['apple', Timedelta('1 days 00:00:00')], dtype=object)
```

In addition to object conversion, `to_numeric()` provides another argument `downcast`, which gives the option of downcasting the newly (or already) numeric data to a smaller dtype, which can conserve memory:

```
In [407]: m = ['1', 2, 3]

In [408]: pd.to_numeric(m, downcast='integer')    # smallest signed int dtype
Out [408]: array([1, 2, 3], dtype=int8)

In [409]: pd.to_numeric(m, downcast='signed')     # same as 'integer'
Out [409]: array([1, 2, 3], dtype=int8)

In [410]: pd.to_numeric(m, downcast='unsigned')   # smallest unsigned int dtype
Out [410]: array([1, 2, 3], dtype=uint8)

In [411]: pd.to_numeric(m, downcast='float')      # smallest float dtype
Out [411]: array([ 1.,  2.,  3.], dtype=float32)
```

As these methods apply only to one-dimensional arrays, lists or scalars; they cannot be used directly on multi-dimensional objects such as DataFrames. However, with `apply()`, we can “apply” the function over each column efficiently:

```
In [412]: import datetime

In [413]: df = pd.DataFrame(['2016-07-09', datetime.datetime(2016, 3, 2)] * 2,
                             dtype='O')

In [414]: df
Out [414]:
           0           1
0  2016-07-09  2016-03-02 00:00:00
1  2016-07-09  2016-03-02 00:00:00

In [415]: df.apply(pd.to_datetime)
Out [415]:
           0           1
0  2016-07-09  2016-03-02
1  2016-07-09  2016-03-02

In [416]: df = pd.DataFrame(['1.1', 2, 3] * 2, dtype='O')

In [417]: df
Out [417]:
           0  1  2
0  1.1  2  3
1  1.1  2  3

In [418]: df.apply(pd.to_numeric)
Out [418]:
           0  1  2
0  1.1  2  3
```

(continues on next page)

(continued from previous page)

```

1  1.1  2  3

In [419]: df = pd.DataFrame([[ '5us', pd.Timedelta('1day')]] * 2, dtype='O')

In [420]: df
Out[420]:
      0      1
0  5us  1 days 00:00:00
1  5us  1 days 00:00:00

In [421]: df.apply(pd.to_timedelta)
Out[421]:
      0      1
0 00:00:00.000005 1 days
1 00:00:00.000005 1 days

```

### 9.13.5 gotchas

Performing selection operations on integer type data can easily upcast the data to `floating`. The dtype of the input data will be preserved in cases where nans are not introduced. See also *Support for integer NA*.

```

In [422]: dfi = df3.astype('int32')

In [423]: dfi['E'] = 1

In [424]: dfi
Out[424]:
   A  B  C  E
0  0  0 255  1
1  0  0 255  1
2  0  0   0  1
3  1  0   0  1
4  0  0   0  1
5  0  0   0  1
6  1 -1   0  1
7  0  0   0  1

In [425]: dfi.dtypes
Out[425]:
A      int32
B      int32
C      int32
E      int64
dtype: object

In [426]: casted = dfi[dfi>0]

In [427]: casted
Out[427]:
   A  B  C  E
0 NaN NaN 255.0  1
1 NaN NaN 255.0  1
2 NaN NaN  NaN  1

```

(continues on next page)

(continued from previous page)

```

3  1.0 NaN      NaN  1
4  NaN NaN      NaN  1
5  NaN NaN      NaN  1
6  1.0 NaN      NaN  1
7  NaN NaN      NaN  1

```

```
In [428]: casted.dtypes
```

```

////////////////////////////////////
↪
A      float64
B      float64
C      float64
E        int64
dtype: object

```

While float dtypes are unchanged.

```
In [429]: dfa = df3.copy()
```

```
In [430]: dfa['A'] = dfa['A'].astype('float32')
```

```
In [431]: dfa.dtypes
```

```
Out[431]:
```

```

A      float32
B      float64
C      float64
dtype: object

```

```
In [432]: casted = dfa[df2>0]
```

```
In [433]: casted
```

```
Out[433]:
```

|   | A        | B        | C     |
|---|----------|----------|-------|
| 0 | NaN      | 0.200071 | 255.0 |
| 1 | NaN      | NaN      | 255.0 |
| 2 | 0.337012 | NaN      | NaN   |
| 3 | 1.483571 | NaN      | NaN   |
| 4 | NaN      | 0.258626 | NaN   |
| 5 | NaN      | 0.941688 | NaN   |
| 6 | NaN      | NaN      | NaN   |
| 7 | NaN      | NaN      | NaN   |

```
In [434]: casted.dtypes
```

```

////////////////////////////////////
↪
A      float32
B      float64
C      float64
dtype: object

```

## 9.14 Selecting columns based on dtype

The `select_dtypes()` method implements subsetting of columns based on their dtype.

First, let's create a *DataFrame* with a slew of different dtypes:

```

In [435]: df = pd.DataFrame({'string': list('abc'),
.....:                     'int64': list(range(1, 4)),
.....:                     'uint8': np.arange(3, 6).astype('u1'),
.....:                     'float64': np.arange(4.0, 7.0),
.....:                     'bool1': [True, False, True],
.....:                     'bool2': [False, True, False],
.....:                     'dates': pd.date_range('now', periods=3).values,
.....:                     'category': pd.Series(list("ABC")).astype('category')})
.....:

In [436]: df['tdeltas'] = df.dates.diff()

In [437]: df['uint64'] = np.arange(3, 6).astype('u8')

In [438]: df['other_dates'] = pd.date_range('20130101', periods=3).values

In [439]: df['tz_aware_dates'] = pd.date_range('20130101', periods=3, tz='US/Eastern')

In [440]: df
Out[440]:
   string  int64  uint8  float64  ...  tdeltas  uint64  other_
↪dates              tz_aware_dates
0      a      1      3      4.0  ...      NaT      3  2013-01-
↪01 2013-01-01 00:00:00-05:00
1      b      2      4      5.0  ...      1 days      4  2013-01-
↪02 2013-01-02 00:00:00-05:00
2      c      3      5      6.0  ...      1 days      5  2013-01-
↪03 2013-01-03 00:00:00-05:00

[3 rows x 12 columns]

```

And the dtypes:

```

In [441]: df.dtypes
Out[441]:
string              object
int64               int64
uint8               uint8
float64             float64
bool1               bool
bool2               bool
dates               datetime64[ns]
category            category
tdeltas             timedelta64[ns]
uint64              uint64
other_dates         datetime64[ns]
tz_aware_dates      datetime64[ns, US/Eastern]
dtype: object

```

`select_dtypes()` has two parameters `include` and `exclude` that allow you to say “give me the columns *with* these dtypes” (include) and/or “give the columns *without* these dtypes” (exclude).

For example, to select `bool` columns:

```

In [442]: df.select_dtypes(include=[bool])
Out[442]:
   bool1  bool2
0    True  False

```

(continues on next page)

(continued from previous page)

```
1 False   True
2   True  False
```

You can also pass the name of a dtype in the [NumPy dtype hierarchy](#):

```
In [443]: df.select_dtypes(include=['bool'])
Out[443]:
   bool1  bool2
0   True  False
1  False   True
2   True  False
```

`select_dtypes()` also works with generic dtypes as well.

For example, to select all numeric and boolean columns while excluding unsigned integers:

```
In [444]: df.select_dtypes(include=['number', 'bool'], exclude=['unsignedinteger'])
Out[444]:
   int64  float64  bool1  bool2  dteltas
0      1      4.0   True  False     NaT
1      2      5.0  False   True  1 days
2      3      6.0   True  False  1 days
```

To select string columns you must use the object dtype:

```
In [445]: df.select_dtypes(include=['object'])
Out[445]:
   string
0      a
1      b
2      c
```

To see all the child dtypes of a generic dtype like `numpy.number` you can define a function that returns a tree of child dtypes:

```
In [446]: def subdtypes(dtype):
.....:     subs = dtype.__subclasses__()
.....:     if not subs:
.....:         return dtype
.....:     return [dtype, [subdtypes(dt) for dt in subs]]
.....:
```

All NumPy dtypes are subclasses of `numpy.generic`:

```
In [447]: subdtypes(np.generic)
Out[447]:
[numpy.generic,
 [ [numpy.number,
    [ [numpy.integer,
      [ [numpy.signedinteger,
        [numpy.int8,
         numpy.int16,
         numpy.int32,
         numpy.int64,
         numpy.int64,
         numpy.timedelta64]],
        [numpy.unsignedinteger,
```

(continues on next page)

(continued from previous page)

```
[numpy.uint8,
 numpy.uint16,
 numpy.uint32,
 numpy.uint64,
 numpy.uint64]]],
[numpy.inexact,
 [[numpy.floating,
  [numpy.float16, numpy.float32, numpy.float64, numpy.float128]],
 [numpy.complexfloating,
  [numpy.complex64, numpy.complex128, numpy.complex256]]]]],
[numpy.flexible,
 [[numpy.character, [numpy.bytes_, numpy.str_]],
 [numpy.void, [numpy.record]]]],
numpy.bool_,
numpy.datetime64,
numpy.object_]]
```

---

**Note:** Pandas also defines the types `category`, and `datetime64[ns, tz]`, which are not integrated into the normal NumPy hierarchy and won't show up with the above function.

---



## WORKING WITH TEXT DATA

Series and Index are equipped with a set of string processing methods that make it easy to operate on each element of the array. Perhaps most importantly, these methods exclude missing/NA values automatically. These are accessed via the `str` attribute and generally have names matching the equivalent (scalar) built-in string methods:

```
In [1]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
```

```
In [2]: s.str.lower()
```

```
Out[2]:
```

```
0      a
1      b
2      c
3    aaba
4    baca
5     NaN
6    caba
7     dog
8     cat
dtype: object
```

```
In [3]: s.str.upper()
```

```
////////////////////////////////////
↪
0      A
1      B
2      C
3    AABA
4    BACA
5     NaN
6    CABA
7     DOG
8     CAT
dtype: object
```

```
In [4]: s.str.len()
```

```
////////////////////////////////////
↪
0      1.0
1      1.0
2      1.0
3      4.0
4      4.0
5     NaN
6      4.0
7      3.0
```

(continues on next page)

(continued from previous page)

```
8      3.0
dtype: float64
```

```
In [5]: idx = pd.Index([' jack', 'jill ', ' jesse ', 'frank'])

In [6]: idx.str.strip()
Out[6]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')

In [7]: idx.str.lstrip()
Out[7]: Index(['jack', 'jill', 'jesse', 'frank'], dtype='object')

In [8]: idx.str.rstrip()
Out[8]: Index([' jack', 'jill ', ' jesse ', 'frank'], dtype='object')
```

The string methods on Index are especially useful for cleaning up or transforming DataFrame columns. For instance, you may have columns with leading or trailing whitespace:

```
In [9]: df = pd.DataFrame(randn(3, 2), columns=[' Column A ', ' Column B '],
...:                      index=range(3))
...:

In [10]: df
Out[10]:
   Column A  Column B
0 -1.425575 -1.336299
1  0.740933  1.032121
2 -1.585660  0.913812
```

Since `df.columns` is an Index object, we can use the `.str` accessor

```
In [11]: df.columns.str.strip()
Out[11]: Index(['Column A', 'Column B'], dtype='object')

In [12]: df.columns.str.lower()
Out[12]: Index(['column a', 'column b'], dtype='object')
```

These string methods can then be used to clean up the columns as needed. Here we are removing leading and trailing whitespaces, lowercasing all names, and replacing any remaining whitespaces with underscores:

```
In [13]: df.columns = df.columns.str.strip().str.lower().str.replace(' ', '_')

In [14]: df
Out[14]:
   column_a  column_b
0 -1.425575 -1.336299
1  0.740933  1.032121
2 -1.585660  0.913812
```

**Note:** If you have a Series where lots of elements are repeated (i.e. the number of unique elements in the Series is a lot smaller than the length of the Series), it can be faster to convert the original Series to one of type category and then use `.str.<method>` or `.dt.<property>` on that. The performance difference comes from the fact that, for Series of type category, the string operations are done on the `.categories` and not on

each element of the Series.

Please note that a Series of type `category` with string `.categories` has some limitations in comparison of Series of type `string` (e.g. you can't add strings to each other: `s + " " + s` won't work if `s` is a Series of type `category`). Also, `.str` methods which operate on elements of type `list` are not available on such a Series.

## 10.1 Splitting and Replacing Strings

Methods like `split` return a Series of lists:

```
In [15]: s2 = pd.Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])

In [16]: s2.str.split('_')
Out[16]:
0      [a, b, c]
1      [c, d, e]
2           NaN
3      [f, g, h]
dtype: object
```

Elements in the split lists can be accessed using `get` or `[]` notation:

```
In [17]: s2.str.split('_').str.get(1)
Out[17]:
0      b
1      d
2     NaN
3      g
dtype: object

In [18]: s2.str.split('_').str[1]
Out[18]:
0      b
1      d
2     NaN
3      g
dtype: object
```

It is easy to expand this to return a DataFrame using `expand`.

```
In [19]: s2.str.split('_', expand=True)
Out[19]:
   0  1  2
0  a  b  c
1  c  d  e
2 NaN NaN NaN
3  f  g  h
```

It is also possible to limit the number of splits:

```
In [20]: s2.str.split('_', expand=True, n=1)
Out[20]:
   0  1
0  a b_c
1  c d_e
```

(continues on next page)

(continued from previous page)

```
2 NaN NaN
3 f g_h
```

`rsplit` is similar to `split` except it works in the reverse direction, i.e., from the end of the string to the beginning of the string:

```
In [21]: s2.str.rsplit('_', expand=True, n=1)
Out[21]:
   0  1
0 a_b  c
1 c_d  e
2 NaN NaN
3 f_g  h
```

`replace` by default replaces regular expressions:

```
In [22]: s3 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca',
.....:                  '', np.nan, 'CABA', 'dog', 'cat'])
.....:
```

```
In [23]: s3
```

```
Out[23]:
0      A
1      B
2      C
3  Aaba
4  Baca
5
6   NaN
7  CABA
8   dog
9   cat
dtype: object
```

```
In [24]: s3.str.replace('^.a|dog', 'XX-XX ', case=False)
```

```

////////////////////////////////////
↪
0      A
1      B
2      C
3  XX-XX ba
4  XX-XX ca
5
6      NaN
7  XX-XX BA
8  XX-XX
9  XX-XX t
dtype: object
```

Some caution must be taken to keep regular expressions in mind! For example, the following code will cause trouble because of the regular expression meaning of `$`:

```
# Consider the following badly formatted financial data
In [25]: dollars = pd.Series(['12', '-$10', '$10,000'])

# This does what you'd naively expect:
```

(continues on next page)

(continued from previous page)

```

In [26]: dollars.str.replace('$', '')
Out[26]:
0      12
1     -10
2    10,000
dtype: object

# But this doesn't:
In [27]: dollars.str.replace('-$', '-')
Out[27]:
0      12
1     -10
2    $10,000
dtype: object

# We need to escape the special character (for >1 len patterns)
In [28]: dollars.str.replace(r'\-$', '-')
Out[28]:
0      12
1     -10
2    $10,000
dtype: object

```

New in version 0.23.0.

If you do want literal replacement of a string (equivalent to `str.replace()`), you can set the optional `regex` parameter to `False`, rather than escaping each character. In this case both `pat` and `repl` must be strings:

```

# These lines are equivalent
In [29]: dollars.str.replace(r'\-$', '-')
Out[29]:
0      12
1     -10
2    $10,000
dtype: object

In [30]: dollars.str.replace('-', '-', regex=False)
Out[30]:
0      12
1     -10
2    $10,000
dtype: object

```

New in version 0.20.0.

The `replace` method can also take a callable as replacement. It is called on every `pat` using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

```

# Reverse every lowercase alphabetic word
In [31]: pat = r'[a-z]+'

In [32]: repl = lambda m: m.group(0)[::-1]

In [33]: pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(pat, repl)
Out[33]:
0    oof 123

```

(continues on next page)

(continued from previous page)

```

1    rab zab
2      NaN
dtype: object

# Using regex groups
In [34]: pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"

In [35]: repl = lambda m: m.group('two').swapcase()

In [36]: pd.Series(['Foo Bar Baz', np.nan]).str.replace(pat, repl)
Out[36]:
0    bAR
1     NaN
dtype: object

```

New in version 0.20.0.

The `replace` method also accepts a compiled regular expression object from `re.compile()` as a pattern. All flags should be included in the compiled regular expression object.

```

In [37]: import re

In [38]: regex_pat = re.compile(r'^.a|dog', flags=re.IGNORECASE)

In [39]: s3.str.replace(regex_pat, 'XX-XX ')
Out[39]:
0          A
1          B
2          C
3    XX-XX ba
4    XX-XX ca
5
6          NaN
7    XX-XX BA
8    XX-XX
9    XX-XX t
dtype: object

```

Including a `flags` argument when calling `replace` with a compiled regular expression object will raise a `ValueError`.

```

In [40]: s3.str.replace(regex_pat, 'XX-XX ', flags=re.IGNORECASE)
-----
ValueError: case and flags cannot be set when pat is a compiled regex

```

## 10.2 Concatenation

There are several ways to concatenate a `Series` or `Index`, either with itself or others, all based on `cat()`, resp. `Index.str.cat`.

### 10.2.1 Concatenating a single Series into a string

The content of a `Series` (or `Index`) can be concatenated:

```
In [41]: s = pd.Series(['a', 'b', 'c', 'd'])
```

```
In [42]: s.str.cat(sep=',')
```

```
Out[42]: 'a,b,c,d'
```

If not specified, the keyword `sep` for the separator defaults to the empty string, `sep=''`:

```
In [43]: s.str.cat()
```

```
Out[43]: 'abcd'
```

By default, missing values are ignored. Using `na_rep`, they can be given a representation:

```
In [44]: t = pd.Series(['a', 'b', np.nan, 'd'])
```

```
In [45]: t.str.cat(sep=',')
```

```
Out[45]: 'a,b,d'
```

```
In [46]: t.str.cat(sep=',', na_rep='-')
```

```
Out[46]: 'a,b,-,d'
```

## 10.2.2 Concatenating a Series and something list-like into a Series

The first argument to `cat()` can be a list-like object, provided that it matches the length of the calling `Series` (or `Index`).

```
In [47]: s.str.cat(['A', 'B', 'C', 'D'])
```

```
Out[47]:
```

```
0    aA
```

```
1    bB
```

```
2    cC
```

```
3    dD
```

```
dtype: object
```

Missing values on either side will result in missing values in the result as well, *unless* `na_rep` is specified:

```
In [48]: s.str.cat(t)
```

```
Out[48]:
```

```
0    aa
```

```
1    bb
```

```
2    NaN
```

```
3    dd
```

```
dtype: object
```

```
In [49]: s.str.cat(t, na_rep='-')
```

```
Out[49]:
```

```
0    aa
```

```
1    bb
```

```
2    c-
```

```
3    dd
```

```
dtype: object
```

## 10.2.3 Concatenating a Series and something array-like into a Series

New in version 0.23.0.

The parameter `others` can also be two-dimensional. In this case, the number of rows must match the lengths of the calling Series (or Index).

```
In [50]: d = pd.concat([t, s], axis=1)

In [51]: s
Out[51]:
0    a
1    b
2    c
3    d
dtype: object

In [52]: d
Out[52]:
0 1
0 a a
1 b b
2 NaN c
3 d d

In [53]: s.str.cat(d, na_rep='-')
Out[53]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object
```

## 10.2.4 Concatenating a Series and an indexed object into a Series, with alignment

New in version 0.23.0.

For concatenation with a Series or DataFrame, it is possible to align the indexes before concatenation by setting the `join-keyword`.

```
In [54]: u = pd.Series(['b', 'd', 'a', 'c'], index=[1, 3, 0, 2])

In [55]: s
Out[55]:
0    a
1    b
2    c
3    d
dtype: object

In [56]: u
Out[56]:
1    b
3    d
0    a
2    c
dtype: object

In [57]: s.str.cat(u)
```

(continues on next page)



(continued from previous page)

```

0    ab
1    bd
2    ca
3    dc
dtype: object

```

```
In [58]: s.str.cat(u, join='left')
```

```

////////////////////////////////////
↪
0    aa
1    bb
2    cc
3    dd
dtype: object

```

**Warning:** If the `join` keyword is not passed, the method `cat()` will currently fall back to the behavior before version 0.23.0 (i.e. no alignment), but a `FutureWarning` will be raised if any of the involved indexes differ, since this default will change to `join='left'` in a future version.

The usual options are available for `join` (one of `'left'`, `'outer'`, `'inner'`, `'right'`). In particular, alignment also means that the different lengths do not need to coincide anymore.

```
In [59]: v = pd.Series(['z', 'a', 'b', 'd', 'e'], index=[-1, 0, 1, 3, 4])
```

```
In [60]: s
```

```

Out[60]:
0    a
1    b
2    c
3    d
dtype: object

```

```
In [61]: v
```

```

////////////////////////////////////Out[61]:
-1    z
0     a
1     b
3     d
4     e
dtype: object

```

```
In [62]: s.str.cat(v, join='left', na_rep='-')
```

```

////////////////////////////////////
↪
0    aa
1    bb
2    c-
3    dd
dtype: object

```

```
In [63]: s.str.cat(v, join='outer', na_rep='-')
```

```

////////////////////////////////////
↪
-1    -z

```

(continues on next page)

(continued from previous page)

```

0    aa
1    bb
2    c-
3    dd
4    -e
dtype: object

```

The same alignment can be used when `others` is a `DataFrame`:

```

In [64]: f = d.loc[[3, 2, 1, 0], :]

In [65]: s
Out[65]:
0    a
1    b
2    c
3    d
dtype: object

In [66]: f
Out[66]:
0 1
3 d d
2 NaN c
1 b b
0 a a

In [67]: s.str.cat(f, join='left', na_rep='-')
Out[67]:
0    aaa
1    bbb
2    c-c
3    ddd
dtype: object

```

## 10.2.5 Concatenating a Series and many objects into a Series

All one-dimensional list-likes can be arbitrarily combined in a list-like container (including iterators, `dict`-views, etc.):

```

In [68]: s
Out[68]:
0    a
1    b
2    c
3    d
dtype: object

In [69]: u
Out[69]:
1    b
3    d
0    a
2    c

```

(continues on next page)



## 10.3 Indexing with `.str`

You can use `[]` notation to directly index by position locations. If you index past the end of the string, the result will be a NaN.

```
In [76]: s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan,
.....:                  'CABA', 'dog', 'cat'])
.....:
```

```
In [77]: s.str[0]
```

```
Out[77]:
```

```
0      A
1      B
2      C
3      A
4      B
5    NaN
6      C
7      d
8      c
dtype: object
```

```
In [78]: s.str[1]
```

```

////////////////////////////////////
↪
0    NaN
1    NaN
2    NaN
3     a
4     a
5    NaN
6     A
7     o
8     a
dtype: object
```

## 10.4 Extracting Substrings

### 10.4.1 Extract first match in each subject (extract)

**Warning:** In version 0.18.0, `extract` gained the `expand` argument. When `expand=False` it returns a `Series`, `Index`, or `DataFrame`, depending on the subject and regular expression pattern (same behavior as pre-0.18.0). When `expand=True` it always returns a `DataFrame`, which is more consistent and less confusing from the perspective of a user. `expand=True` is the default since version 0.23.0.

The `extract` method accepts a [regular expression](#) with at least one capture group.

Extracting a regular expression with more than one group returns a `DataFrame` with one column per group.

```
In [79]: pd.Series(['a1', 'b2', 'c3']).str.extract('([ab])(\d)', expand=False)
Out[79]:
   0  1
```

(continues on next page)

(continued from previous page)

```
0    a    1
1    b    2
2  NaN  NaN
```

Elements that do not match return a row filled with NaN. Thus, a Series of messy strings can be “converted” into a like-indexed Series or DataFrame of cleaned-up or more useful strings, without necessitating `get()` to access tuples or `re.match` objects. The dtype of the result is always object, even if no match is found and the result only contains NaN.

Named groups like

```
In [80]: pd.Series(['a1', 'b2', 'c3']).str.extract('(P<letter>[ab])(P<digit>\d)',
↳ expand=False)
Out [80]:
   letter digit
0      a      1
1      b      2
2    NaN    NaN
```

and optional groups like

```
In [81]: pd.Series(['a1', 'b2', '3']).str.extract('([ab])?(\d)', expand=False)
Out [81]:
   0  1
0   a  1
1   b  2
2  NaN 3
```

can also be used. Note that any capture group names in the regular expression will be used for column names; otherwise capture group numbers will be used.

Extracting a regular expression with one group returns a DataFrame with one column if `expand=True`.

```
In [82]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=True)
Out [82]:
   0
0   1
1   2
2  NaN
```

It returns a Series if `expand=False`.

```
In [83]: pd.Series(['a1', 'b2', 'c3']).str.extract('[ab](\d)', expand=False)
Out [83]:
0      1
1      2
2    NaN
dtype: object
```

Calling on an Index with a regex with exactly one capture group returns a DataFrame with one column if `expand=True`.

```
In [84]: s = pd.Series(['a1', 'b2', 'c3'], ['A11', 'B22', 'C33'])

In [85]: s
Out [85]:
A11    a1
```

(continues on next page)

(continued from previous page)

```

B22    b2
C33    c3
dtype: object

In [86]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=True)
\\Out [86]:
  letter
0      A
1      B
2      C

```

It returns an Index if `expand=False`.

```

In [87]: s.index.str.extract("(?P<letter>[a-zA-Z])", expand=False)
Out [87]: Index(['A', 'B', 'C'], dtype='object', name='letter')

```

Calling on an Index with a regex with more than one capture group returns a DataFrame if `expand=True`.

```

In [88]: s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=True)
Out [88]:
  letter  1
0      A  11
1      B  22
2      C  33

```

It raises `ValueError` if `expand=False`.

```

>>> s.index.str.extract("(?P<letter>[a-zA-Z])([0-9]+)", expand=False)
ValueError: only one regex group is supported with Index

```

The table below summarizes the behavior of `extract` (`expand=False`) (input subject in first column, number of groups in regex in first row)

|        | 1 group | >1 group   |
|--------|---------|------------|
| Index  | Index   | ValueError |
| Series | Series  | DataFrame  |

## 10.4.2 Extract all matches in each subject (extractall)

New in version 0.18.0.

Unlike `extract` (which returns only the first match),

```

In [89]: s = pd.Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])

In [90]: s
Out [90]:
A    a1a2
B     b1
C     c1
dtype: object

In [91]: two_groups = '(?P<letter>[a-z])(?P<digit>[0-9])'

```

(continues on next page)

(continued from previous page)

```
In [92]: s.str.extract(two_groups, expand=True)
```

Out [92] :

|   | letter | digit |
|---|--------|-------|
| A | a      | 1     |
| B | b      | 1     |
| C | c      | 1     |

the `extractall` method returns every match. The result of `extractall` is always a `DataFrame` with a `MultiIndex` on its rows. The last level of the `MultiIndex` is named `match` and indicates the order in the subject.

```
In [93]: s.str.extractall(two_groups)
```

Out [93] :

|   |       | letter | digit |
|---|-------|--------|-------|
|   | match |        |       |
| A | 0     | a      | 1     |
|   | 1     | a      | 2     |
| B | 0     | b      | 1     |
| C | 0     | c      | 1     |

When each subject string in the Series has exactly one match,

```
In [94]: s = pd.Series(['a3', 'b3', 'c2'])
```

In [95]: s

Out [95] :

```
0      a3
1      b3
2      c2
dtype: object
```

then `extractall(pat).xs(0, level='match')` gives the same result as `extract(pat)`.

```
In [96]: extract_result = s.str.extract(two_groups, expand=True)
```

```
In [97]: extract_result
```

Out [97] :

```

letter digit
0      a      3
1      b      3
2      c      2

```

```
In [98]: extractall_result = s.str.extractall(two_groups)
```

```
In [99]: extractall_result
```

Out [99] :

|       |   | letter | digit |
|-------|---|--------|-------|
| match |   |        |       |
| 0     | 0 | a      | 3     |
| 1     | 0 | b      | 3     |
| 2     | 0 | c      | 2     |

```
In [100]: extractall_result.xs(0, level="match")
```

(continues on next page)

(continued from previous page)

|   |   |   |
|---|---|---|
| 0 | a | 3 |
| 1 | b | 3 |
| 2 | c | 2 |

Index also supports `.str.extractall`. It returns a `DataFrame` which has the same result as a `Series.str.extractall` with a default index (starts from 0).

New in version 0.19.0.

```
In [101]: pd.Index(["a1a2", "b1", "c1"]).str.extractall(two_groups)
```

```
Out[101]:
```

|   |       | letter | digit |
|---|-------|--------|-------|
|   | match |        |       |
| 0 | 0     | a      | 1     |
|   | 1     | a      | 2     |
| 1 | 0     | b      | 1     |
| 2 | 0     | c      | 1     |

```
In [102]: pd.Series(["a1a2", "b1", "c1"]).str.extractall(two_groups)
```

```

////////////////////////////////////
↪
```

|   |       | letter | digit |
|---|-------|--------|-------|
|   | match |        |       |
| 0 | 0     | a      | 1     |
|   | 1     | a      | 2     |
| 1 | 0     | b      | 1     |
| 2 | 0     | c      | 1     |

## 10.5 Testing for Strings that Match or Contain a Pattern

You can check whether elements contain a pattern:

```
In [103]: pattern = r'[0-9][a-z]'
```

```
In [104]: pd.Series(['1', '2', '3a', '3b', '03c']).str.contains(pattern)
```

```
Out[104]:
```

```

0    False
1    False
2     True
3     True
4     True
dtype: bool
```

Or whether elements match a pattern:

```
In [105]: pd.Series(['1', '2', '3a', '3b', '03c']).str.match(pattern)
```

```
Out[105]:
```

```

0    False
1    False
2     True
3     True
4    False
dtype: bool
```



The distinction between `match` and `contains` is strictness: `match` relies on strict `re.match`, while `contains` relies on `re.search`.

Methods like `match`, `contains`, `startswith`, and `endswith` take an extra `na` argument so missing values can be considered `True` or `False`:

```
In [106]: s4 = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat',
→])

In [107]: s4.str.contains('A', na=False)
Out[107]:
0      True
1     False
2     False
3      True
4     False
5     False
6      True
7     False
8     False
dtype: bool
```

## 10.6 Creating Indicator Variables

You can extract dummy variables from string columns. For example if they are separated by a `'|'`:

```
In [108]: s = pd.Series(['a', 'a|b', np.nan, 'a|c'])

In [109]: s.str.get_dummies(sep='|')
Out[109]:
   a  b  c
0  1  0  0
1  1  1  0
2  0  0  0
3  1  0  1
```

String Index also supports `get_dummies` which returns a `MultiIndex`.

New in version 0.18.1.

```
In [110]: idx = pd.Index(['a', 'a|b', np.nan, 'a|c'])

In [111]: idx.str.get_dummies(sep='|')
Out[111]:
MultiIndex(levels=[[0, 1], [0, 1], [0, 1]],
            labels=[[1, 1, 0, 1], [0, 1, 0, 0], [0, 0, 0, 1]],
            names=['a', 'b', 'c'])
```

See also `get_dummies()`.

## 10.7 Method Summary

| Method                       | Description  |
|------------------------------|--|
| <code>cat()</code>           | Concatenate strings  |
| <code>split()</code>         | Split strings on delimiter   |
| <code>rsplit()</code>        | Split strings on delimiter working from the end of the string  |
| <code>get()</code>           | Index into each element (retrieve i-th element)  |
| <code>join()</code>          | Join strings in each element of the Series with passed separator   |
| <code>get_dummies()</code>   | Split strings on the delimiter returning DataFrame of dummy variables  |
| <code>contains()</code>      | Return boolean array if each string contains pattern/regex   |
| <code>replace()</code>       | Replace occurrences of pattern/regex/string with some other string or the return value of a callable given the occurrence                  |
| <code>repeat()</code>        | Duplicate values ( <code>s.str.repeat(3)</code> equivalent to <code>x * 3</code> )   |
| <code>pad()</code>           | Add whitespace to left, right, or both sides of strings  |
| <code>center()</code>        | Equivalent to <code>str.center</code>  |
| <code>ljust()</code>         | Equivalent to <code>str.ljust</code>   |
| <code>rjust()</code>         | Equivalent to <code>str.rjust</code>   |
| <code>zfill()</code>         | Equivalent to <code>str.zfill</code>   |
| <code>wrap()</code>          | Split long strings into lines with length less than a given width  |
| <code>slice()</code>         | Slice each string in the Series  |
| <code>slice_replace()</code> | Replace slice in each string with passed value   |
| <code>count()</code>         | Count occurrences of pattern   |
| <code>startswith()</code>    | Equivalent to <code>str.startswith(pat)</code> for each element  |
| <code>endswith()</code>      | Equivalent to <code>str.endswith(pat)</code> for each element  |
| <code>findall()</code>       | Compute list of all occurrences of pattern/regex for each string   |
| <code>match()</code>         | Call <code>re.match</code> on each element, returning matched groups as list   |
| <code>extract()</code>       | Call <code>re.search</code> on each element, returning DataFrame with one row for each element and one column for each regex capture group |
| <code>extractall()</code>    | Call <code>re.findall</code> on each element, returning DataFrame with one row for each match and one column for each regex capture group  |
| <code>len()</code>           | Compute string lengths   |
| <code>strip()</code>         | Equivalent to <code>str.strip</code>   |
| <code>rstrip()</code>        | Equivalent to <code>str.rstrip</code>  |
| <code>lstrip()</code>        | Equivalent to <code>str.lstrip</code>  |
| <code>partition()</code>     | Equivalent to <code>str.partition</code>   |
| <code>rpartition()</code>    | Equivalent to <code>str.rpartition</code>  |
| <code>lower()</code>         | Equivalent to <code>str.lower</code>   |
| <code>upper()</code>         | Equivalent to <code>str.upper</code>   |
| <code>find()</code>          | Equivalent to <code>str.find</code>  |
| <code>rfind()</code>         | Equivalent to <code>str.rfind</code>   |
| <code>index()</code>         | Equivalent to <code>str.index</code>   |
| <code>rindex()</code>        | Equivalent to <code>str.rindex</code>  |
| <code>capitalize()</code>    | Equivalent to <code>str.capitalize</code>  |
| <code>swapcase()</code>      | Equivalent to <code>str.swapcase</code>  |
| <code>normalize()</code>     | Return Unicode normal form. Equivalent to <code>unicodedata.normalize</code>   |
| <code>translate()</code>     | Equivalent to <code>str.translate</code>   |
| <code>isalnum()</code>       | Equivalent to <code>str.isalnum</code>   |
| <code>isalpha()</code>       | Equivalent to <code>str.isalpha</code>   |
| <code>isdigit()</code>       | Equivalent to <code>str.isdigit</code>   |
| <code>isspace()</code>       | Equivalent to <code>str.isspace</code>   |
| <code>islower()</code>       | Equivalent to <code>str.islower</code>   |
| <code>isupper()</code>       | Equivalent to <code>str.isupper</code>   |
| <code>istitle()</code>       | Equivalent to <code>str.istitle</code>   |

Continued on next page

Table 1 – continued from previous page

| Method                   | Description                              |
|--------------------------|--|
| <code>isnumeric()</code> | Equivalent to <code>str.isnumeric</code> |
| <code>isdecimal()</code> | Equivalent to <code>str.isdecimal</code> |



## OPTIONS AND SETTINGS

### 11.1 Overview

pandas has an options system that lets you customize some aspects of its behaviour, display-related options being those the user is most likely to adjust.

Options have a full “dotted-style”, case-insensitive name (e.g. `display.max_rows`). You can get/set options directly as attributes of the top-level `options` attribute:

```
In [1]: import pandas as pd

In [2]: pd.options.display.max_rows
Out[2]: 15

In [3]: pd.options.display.max_rows = 999

In [4]: pd.options.display.max_rows
Out[4]: 999
```

The API is composed of 5 relevant functions, available directly from the `pandas` namespace:

- `get_option()` / `set_option()` - get/set the value of a single option.
- `reset_option()` - reset one or more options to their default value.
- `describe_option()` - print the descriptions of one or more options.
- `option_context()` - execute a codeblock with a set of options that revert to prior settings after execution.

**Note:** Developers can check out [pandas/core/config.py](#) for more information.

All of the functions above accept a regexp pattern (`re.search` style) as an argument, and so passing in a substring will work - as long as it is unambiguous:

```
In [5]: pd.get_option("display.max_rows")
Out[5]: 999

In [6]: pd.set_option("display.max_rows", 101)

In [7]: pd.get_option("display.max_rows")
Out[7]: 101

In [8]: pd.set_option("max_r", 102)

In [9]: pd.get_option("display.max_rows")
Out[9]: 102
```

The following will **not work** because it matches multiple option names, e.g. `display.max_colwidth`, `display.max_rows`, `display.max_columns`:

```
In [10]: try:
.....:     pd.get_option("column")
.....: except KeyError as e:
.....:     print(e)
.....:
'Pattern matched multiple keys'
```

**Note:** Using this form of shorthand may cause your code to break if new options with similar names are added in future versions.

You can get a list of available options and their descriptions with `describe_option`. When called with no argument `describe_option` will print out the descriptions for all available options.

## 11.2 Getting and Setting Options

As described above, `get_option()` and `set_option()` are available from the pandas namespace. To change an option, call `set_option('option regex', new_value)`.

```
In [11]: pd.get_option('mode.sim_interactive')
Out[11]: False

In [12]: pd.set_option('mode.sim_interactive', True)

In [13]: pd.get_option('mode.sim_interactive')
Out[13]: True
```

**Note:** The option ‘mode.sim\_interactive’ is mostly used for debugging purposes.

All options also have a default value, and you can use `reset_option` to do just that:

```
In [14]: pd.get_option("display.max_rows")
Out[14]: 60

In [15]: pd.set_option("display.max_rows", 999)

In [16]: pd.get_option("display.max_rows")
Out[16]: 999

In [17]: pd.reset_option("display.max_rows")

In [18]: pd.get_option("display.max_rows")
Out[18]: 60
```

It’s also possible to reset multiple options at once (using a regex):

```
In [19]: pd.reset_option("^display")
```

`option_context` context manager has been exposed through the top-level API, allowing you to execute code with given option values. Option values are restored automatically when you exit the *with* block:

```
In [20]: with pd.option_context("display.max_rows", 10, "display.max_columns", 5):
.....:     print(pd.get_option("display.max_rows"))
.....:     print(pd.get_option("display.max_columns"))
```

(continues on next page)

(continued from previous page)

```

.....:
10
5

In [21]: print(pd.get_option("display.max_rows"))
\\\\"60

In [22]: print(pd.get_option("display.max_columns"))
\\\\"0

```

## 11.3 Setting Startup Options in python/ipython Environment

Using startup scripts for the python/ipython environment to import pandas and set options makes working with pandas more efficient. To do this, create a .py or .ipy script in the startup directory of the desired profile. An example where the startup folder is in a default ipython profile can be found at:

```
$IPYTHONDIR/profile_default/startup
```

More information can be found in the [ipython documentation](#). An example startup script for pandas is displayed below:

```

import pandas as pd
pd.set_option('display.max_rows', 999)
pd.set_option('precision', 5)

```

## 11.4 Frequently Used Options

The following is a walkthrough of the more frequently used display options.

`display.max_rows` and `display.max_columns` sets the maximum number of rows and columns displayed when a frame is pretty-printed. Truncated lines are replaced by an ellipsis.

```

In [23]: df = pd.DataFrame(np.random.randn(7,2))

In [24]: pd.set_option('max_rows', 7)

In [25]: df
Out[25]:
      0         1
0  0.469112 -0.282863
1 -1.509059 -1.135632
2  1.212112 -0.173215
3  0.119209 -1.044236
4 -0.861849 -2.104569
5 -0.494929  1.071804
6  0.721555 -0.706771

In [26]: pd.set_option('max_rows', 5)

In [27]: df
Out[27]:

```

(continues on next page)

(continued from previous page)

```

      0      1
0  0.469112 -0.282863
1 -1.509059 -1.135632
..      ...      ...
5 -0.494929  1.071804
6  0.721555 -0.706771

```

```
[7 rows x 2 columns]
```

```
In [28]: pd.reset_option('max_rows')
```

`display.expand_frame_repr` allows for the representation of dataframes to stretch across pages, wrapped over the full column vs row-wise.

```
In [29]: df = pd.DataFrame(np.random.randn(5,10))
```

```
In [30]: pd.set_option('expand_frame_repr', True)
```

```
In [31]: df
```

```
Out[31]:
```

```

      0      1      2      3      4      5      6      7      8      9
0 -1.039575  0.271860 -0.424972  0.567020  0.276232 -1.087401 -0.673690  0.113648 -1.478427  0.524988
1  0.404705  0.577046 -1.715002 -1.039268 -0.370647 -1.157892 -1.344312  0.844885  1.075770 -0.109050
2  1.643563 -1.469388  0.357021 -0.674600 -1.776904 -0.968914 -1.294524  0.413738  0.276662 -0.472035
3 -0.013960 -0.362543 -0.006154 -0.923061  0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309
4 -1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127 -1.436737

```

```
In [32]: pd.set_option('expand_frame_repr', False)
```

```
In [33]: df
```

```
Out[33]:
```

```

      0      1      2      3      4      5      6      7      8      9
0 -1.039575  0.271860 -0.424972  0.567020  0.276232 -1.087401 -0.673690  0.113648 -1.478427  0.524988
1  0.404705  0.577046 -1.715002 -1.039268 -0.370647 -1.157892 -1.344312  0.844885  1.075770 -0.109050
2  1.643563 -1.469388  0.357021 -0.674600 -1.776904 -0.968914 -1.294524  0.413738  0.276662 -0.472035
3 -0.013960 -0.362543 -0.006154 -0.923061  0.895717  0.805244 -1.206412  2.565646  1.431256  1.340309
4 -1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317 -0.076467 -1.187678  1.130127 -1.436737

```

```
In [34]: pd.reset_option('expand_frame_repr')
```

`display.large_repr` lets you select whether to display dataframes that exceed `max_columns` or `max_rows` as a truncated frame, or as a summary.

```
In [35]: df = pd.DataFrame(np.random.randn(10,10))
```

(continues on next page)



(continued from previous page)

```

In [36]: pd.set_option('max_rows', 5)

In [37]: pd.set_option('large_repr', 'truncate')

In [38]: df
Out[38]:
      0         1         2         3         4         5         6         7
↪  8      9
0 -1.413681  1.607920  1.024180  0.569605  0.875906 -2.211372  0.974466 -2.006747 -0.
↪ 410001 -0.078638
1  0.545952 -1.219217 -1.226825  0.769804 -1.281247 -0.727707 -0.121306 -0.097883 0.
↪ 695775  0.341734
..      ...      ...      ...      ...      ...      ...      ...      ...
↪      ...      ...
8 -2.484478 -0.281461  0.030711  0.109121  1.126203 -0.977349  1.474071 -0.064034 -1.
↪ 282782  0.781836
9 -1.071357  0.441153  2.353925  0.583787  0.221471 -0.744471  0.758527  1.729689 -0.
↪ 964980 -0.845696

[10 rows x 10 columns]

In [39]: pd.set_option('large_repr', 'info')

In [40]: df
Out[40]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0      10 non-null float64
1      10 non-null float64
2      10 non-null float64
3      10 non-null float64
4      10 non-null float64
5      10 non-null float64
6      10 non-null float64
7      10 non-null float64
8      10 non-null float64
9      10 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [41]: pd.reset_option('large_repr')

In [42]: pd.reset_option('max_rows')

```

`display.max_colwidth` sets the maximum width of columns. Cells of this length or longer will be truncated with an ellipsis.

```

In [43]: df = pd.DataFrame(np.array([['foo', 'bar', 'bim', 'uncomfortably long string
↪'],
      ....:                        ['horse', 'cow', 'banana', 'apple']]))
      ....:

In [44]: pd.set_option('max_colwidth', 40)

```

(continues on next page)

(continued from previous page)

```

In [45]: df
Out[45]:
   0    1    2    3
0  foo bar  bim uncomfortably long string
1 horse cow banana                apple

In [46]: pd.set_option('max_colwidth', 6)

In [47]: df
Out[47]:
   0    1    2    3
0  foo bar  bim un...
1 horse cow ba... apple

In [48]: pd.reset_option('max_colwidth')

```

`display.max_info_columns` sets a threshold for when by-column info will be given.

```

In [49]: df = pd.DataFrame(np.random.randn(10,10))

In [50]: pd.set_option('max_info_columns', 11)

In [51]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0      10 non-null float64
1      10 non-null float64
2      10 non-null float64
3      10 non-null float64
4      10 non-null float64
5      10 non-null float64
6      10 non-null float64
7      10 non-null float64
8      10 non-null float64
9      10 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes

In [52]: pd.set_option('max_info_columns', 5)

In [53]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Columns: 10 entries, 0 to 9
dtypes: float64(10)
memory usage: 880.0 bytes

In [54]: pd.reset_option('max_info_columns')

```

`display.max_info_rows: df.info()` will usually show null-counts for each column. For large frames this can be quite slow. `max_info_rows` and `max_info_cols` limit this null check only to frames with smaller dimensions than specified. Note that you can specify the option `df.info(null_counts=True)` to override on showing a particular frame.

```

In [55]: df = pd.DataFrame(np.random.choice([0,1,np.nan], size=(10,10)))

```

(continues on next page)

(continued from previous page)

```
In [56]: df
Out[56]:
```

|   | 0   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0.0 | 1.0 | 1.0 | 0.0 | 1.0 | 1.0 | 0.0 | NaN | 1.0 | NaN |
| 1 | 1.0 | NaN | 0.0 | 0.0 | 1.0 | 1.0 | NaN | 1.0 | 0.0 | 1.0 |
| 2 | NaN | NaN | NaN | 1.0 | 1.0 | 0.0 | NaN | 0.0 | 1.0 | NaN |
| 3 | 0.0 | 1.0 | 1.0 | NaN | 0.0 | NaN | 1.0 | NaN | NaN | 0.0 |
| 4 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | NaN | 0.0 | 0.0 |
| 5 | 0.0 | NaN | 1.0 | NaN | NaN | NaN | NaN | 0.0 | 1.0 | NaN |
| 6 | 0.0 | 1.0 | 0.0 | 0.0 | NaN | 1.0 | NaN | NaN | 0.0 | NaN |
| 7 | 0.0 | NaN | 1.0 | 1.0 | NaN | 1.0 | 1.0 | 1.0 | 1.0 | NaN |
| 8 | 0.0 | 0.0 | NaN | 0.0 | NaN | 1.0 | 0.0 | 0.0 | NaN | NaN |
| 9 | NaN | NaN | 0.0 | NaN | NaN | NaN | 0.0 | 1.0 | 1.0 | NaN |

```
In [57]: pd.set_option('max_info_rows', 11)
```

```
In [58]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0      8 non-null float64
1      5 non-null float64
2      8 non-null float64
3      7 non-null float64
4      5 non-null float64
5      7 non-null float64
6      6 non-null float64
7      6 non-null float64
8      8 non-null float64
9      3 non-null float64
dtypes: float64(10)
memory usage: 880.0 bytes
```

```
In [59]: pd.set_option('max_info_rows', 5)
```

```
In [60]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 10 columns):
0      float64
1      float64
2      float64
3      float64
4      float64
5      float64
6      float64
7      float64
8      float64
9      float64
dtypes: float64(10)
memory usage: 880.0 bytes
```

```
In [61]: pd.reset_option('max_info_rows')
```

`display.precision` sets the output display precision in terms of decimal places. This is only a suggestion.

```
In [62]: df = pd.DataFrame(np.random.randn(5,5))

In [63]: pd.set_option('precision',7)

In [64]: df
Out[64]:
```

|   | 0          | 1          | 2          | 3          | 4         |
|---|------------|------------|------------|------------|-----------|
| 0 | -2.0490276 | 2.8466122  | -1.2080493 | -0.4503923 | 2.4239054 |
| 1 | 0.1211080  | 0.2669165  | 0.8438259  | -0.2225400 | 2.0219807 |
| 2 | -0.7167894 | -2.2244851 | -1.0611370 | -0.2328247 | 0.4307933 |
| 3 | -0.6654779 | 1.8298075  | -1.4065093 | 1.0782481  | 0.3227741 |
| 4 | 0.2003243  | 0.8900241  | 0.1948132  | 0.3516326  | 0.4488815 |

```
In [65]: pd.set_option('precision',4)

In [66]: df
Out[66]:
```

|   | 0       | 1       | 2       | 3       | 4      |
|---|---------|---------|---------|---------|--------|
| 0 | -2.0490 | 2.8466  | -1.2080 | -0.4504 | 2.4239 |
| 1 | 0.1211  | 0.2669  | 0.8438  | -0.2225 | 2.0220 |
| 2 | -0.7168 | -2.2245 | -1.0611 | -0.2328 | 0.4308 |
| 3 | -0.6655 | 1.8298  | -1.4065 | 1.0782  | 0.3228 |
| 4 | 0.2003  | 0.8900  | 0.1948  | 0.3516  | 0.4489 |

`display.chop_threshold` sets at what level pandas rounds to zero when it displays a Series of DataFrame. This setting does not change the precision at which the number is stored.

```
In [67]: df = pd.DataFrame(np.random.randn(6,6))

In [68]: pd.set_option('chop_threshold', 0)

In [69]: df
Out[69]:
```

|   | 0       | 1       | 2       | 3       | 4       | 5       |
|---|---------|---------|---------|---------|---------|---------|
| 0 | -0.1979 | 0.9657  | -1.5229 | -0.1166 | 0.2956  | -1.0477 |
| 1 | 1.6406  | 1.9058  | 2.7721  | 0.0888  | -1.1442 | -0.6334 |
| 2 | 0.9254  | -0.0064 | -0.8204 | -0.6009 | -1.0393 | 0.8248  |
| 3 | -0.8241 | -0.3377 | -0.9278 | -0.8401 | 0.2485  | -0.1093 |
| 4 | 0.4320  | -0.4607 | 0.3365  | -3.2076 | -1.5359 | 0.4098  |
| 5 | -0.6731 | -0.7411 | -0.1109 | -2.6729 | 0.8645  | 0.0609  |

```
In [70]: pd.set_option('chop_threshold', .5)

In [71]: df
Out[71]:
```

|   | 0       | 1       | 2       | 3       | 4       | 5       |
|---|---------|---------|---------|---------|---------|---------|
| 0 | 0.0000  | 0.9657  | -1.5229 | 0.0000  | 0.0000  | -1.0477 |
| 1 | 1.6406  | 1.9058  | 2.7721  | 0.0000  | -1.1442 | -0.6334 |
| 2 | 0.9254  | 0.0000  | -0.8204 | -0.6009 | -1.0393 | 0.8248  |
| 3 | -0.8241 | 0.0000  | -0.9278 | -0.8401 | 0.0000  | 0.0000  |
| 4 | 0.0000  | 0.0000  | 0.0000  | -3.2076 | -1.5359 | 0.0000  |
| 5 | -0.6731 | -0.7411 | 0.0000  | -2.6729 | 0.8645  | 0.0000  |

```
In [72]: pd.reset_option('chop_threshold')
```

`display.colheader_justify` controls the justification of the headers. The options are 'right', and 'left'.

```

In [73]: df = pd.DataFrame(np.array([np.random.randn(6), np.random.randint(1,9,6)*.1,
↳ np.zeros(6)]).T,
    ....:                  columns=['A', 'B', 'C'], dtype='float')
    ....:

In [74]: pd.set_option('colheader_justify', 'right')

In [75]: df
Out[75]:
      A      B      C
0  0.9331  0.3  0.0
1  0.2888  0.2  0.0
2  1.3250  0.2  0.0
3  0.5892  0.7  0.0
4  0.5314  0.1  0.0
5 -1.1987  0.7  0.0

In [76]: pd.set_option('colheader_justify', 'left')

In [77]: df
Out[77]:
      A      B      C
0  0.9331  0.3  0.0
1  0.2888  0.2  0.0
2  1.3250  0.2  0.0
3  0.5892  0.7  0.0
4  0.5314  0.1  0.0
5 -1.1987  0.7  0.0

In [78]: pd.reset_option('colheader_justify')

```

## 11.5 Available Options

| Option                           | Default  | Function  |
|----------------------------------|----------|---|
| display.chop_threshold           | None     | If set to a float value, all float values smaller then the given threshold will be displayed.   |
| display.colheader_justify        | right    | Controls the justification of column headers. used by DataFrameFormatter.                       |
| display.column_space             | 12       | No description available.   |
| display.date_dayfirst            | False    | When True, prints and parses dates with the day first, eg 20/01/2005                            |
| display.date_yearfirst           | False    | When True, prints and parses dates with the year first, eg 2005/01/20                           |
| display.encoding                 | UTF-8    | Defaults to the detected encoding of the console. Specifies the encoding to be used.            |
| display.expand_frame_repr        | True     | Whether to print out the full DataFrame repr for wide DataFrames across multiple lines.         |
| display.float_format             | None     | The callable should accept a floating point number and return a string with the desired format. |
| display.large_repr               | truncate | For DataFrames exceeding max_rows/max_cols, the repr (and HTML repr) can be truncated.          |
| display.latex.repr               | False    | Whether to produce a latex DataFrame representation for jupyter frontends that support it.      |
| display.latex.escape             | True     | Escapes special characters in DataFrames, when using the to_latex method.                       |
| display.latex.longtable          | False    | Specifies if the to_latex method of a DataFrame uses the longtable format.                      |
| display.latex.multicolumn        | True     | Combines columns when using a MultiIndex  |
| display.latex.multicolumn_format | 'l'      | Alignment of multicolumn labels   |
| display.latex.multirow           | False    | Combines rows when using a MultiIndex. Centered instead of top-aligned, separated by &#x2013;   |
| display.max_columns              | 0 or 20  | max_rows and max_columns are used in __repr__() methods to decide if to truncate.               |
| display.max_colwidth             | 50       | The maximum width in characters of a column in the repr of a pandas data structure.             |

| Option                                  | Default  | Function  |
|---|----------|---|
| display.max_info_columns                | 100      | max_info_columns is used in DataFrame.info method to decide if per column info        |
| display.max_info_rows                   | 1690785  | df.info() will usually show null-counts for each column. For large frames this ca     |
| display.max_rows                        | 60       | This sets the maximum number of rows pandas should output when printing out           |
| display.max_seq_items                   | 100      | when pretty-printing a long sequence, no more then <i>max_seq_items</i> will be print |
| display.memory_usage                    | True     | This specifies if the memory usage of a DataFrame should be displayed when th         |
| display.multi_sparse                    | True     | “Sparsify” MultiIndex display (don’t display repeated elements in outer levels w      |
| display.notebook_repr_html              | True     | When True, IPython notebook will use html representation for pandas objects (if       |
| display.pprint_nest_depth               | 3        | Controls the number of nested levels to process when pretty-printing                  |
| display.precision                       | 6        | Floating point output precision in terms of number of places after the decimal, fo    |
| display.show_dimensions                 | truncate | Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is sp     |
| display.width                           | 80       | Width of the display in characters. In case python/IPython is running in a termin     |
| display.html.table_schema               | False    | Whether to publish a Table Schema representation for frontends that support it.       |
| display.html.border                     | 1        | A border=value attribute is inserted in the <table> tag for the DataFrame             |
| display.html.use_mathjax                | True     | When True, Jupyter notebook will process table contents using MathJax, renderi        |
| io.excel.xls.writer                     | xlwt     | The default Excel writer engine for ‘xls’ files.                                      |
| io.excel.xlsm.writer                    | openpyxl | The default Excel writer engine for ‘xlsm’ files. Available options: ‘openpyxl’ (t    |
| io.excel.xlsx.writer                    | openpyxl | The default Excel writer engine for ‘xlsx’ files.                                     |
| io.hdf.default_format                   | None     | default format writing format, if None, then put will default to ‘fixed’ and appen    |
| io.hdf.dropna_table                     | True     | drop ALL nan rows when appending to a table   |
| io.parquet.engine                       | None     | The engine to use as a default for parquet reading and writing. If None then try ‘    |
| mode.chained_assignment                 | warn     | Controls SettingWithCopyWarning: ‘raise’, ‘warn’, or None. Raise an ex                |
| mode.sim_interactive                    | False    | Whether to simulate interactive mode for purposes of testing.                         |
| mode.use_inf_as_na                      | False    | True means treat None, NaN, -INF, INF as NA (old way), False means None and           |
| compute.use_bottleneck                  | True     | Use the bottleneck library to accelerate computation if it is installed.              |
| compute.use_numexpr                     | True     | Use the numexpr library to accelerate computation if it is installed.                 |
| plotting.matplotlib.register_converters | True     | Register custom converters with matplotlib. Set to False to de-register.              |

## 11.6 Number Formatting

pandas also allows you to set how numbers are displayed in the console. This option is not set through the `set_options` API.

Use the `set_eng_float_format` function to alter the floating-point formatting of pandas objects to produce a particular format.

For instance:

```
In [79]: import numpy as np

In [80]: pd.set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [81]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [82]: s/1.e3
Out[82]:
a    -236.866u
b     846.974u
c    -685.597u
d     609.099u
e    -303.961u
dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [83]: s/1.e6
```

```

////////////////////////////////////Out [83]:
↪
a    -236.866n
b     846.974n
c    -685.597n
d     609.099n
e    -303.961n
dtype: float64

```

To round floats on a case-by-case basis, you can also use `round()` and `round()`.

## 11.7 Unicode Formatting

**Warning:** Enabling this option will affect the performance for printing of DataFrame and Series (about 2 times slower). Use only when it is actually required.

Some East Asian countries use Unicode characters whose width corresponds to two Latin characters. If a DataFrame or Series contains these characters, the default output mode may not align them properly.

**Note:** Screen captures are attached for each output to show the actual results.

```
In [84]: df = pd.DataFrame({'u': ['UK', u''], u': ['Alice', u'']})
```

```
In [85]: df;
```

```

>>> df = pd.DataFrame({'u'国籍': ['UK', u'日本'], u'名前': ['Alice', u'しのぶ']})
>>> df
   名前  国籍
0  Alice  UK
1  のぶ  日本

```

Enabling `display.unicode.east_asian_width` allows pandas to check each character’s “East Asian Width” property. These characters can be aligned properly by setting this option to `True`. However, this will result in longer render times than the standard `len` function.

```
In [86]: pd.set_option('display.unicode.east_asian_width', True)
```

```
In [87]: df;
```

```

>>> pd.set_option('display.unicode.east_asian_width', True)
>>> df
   名前  国籍
0  Alice  UK
1  のぶ  日本

```

In addition, Unicode characters whose width is “Ambiguous” can either be 1 or 2 characters wide depending on the terminal setting or encoding. The option `display.unicode.ambiguous_as_wide` can be used to handle the ambiguity.

By default, an “Ambiguous” character’s width, such as “¡” (inverted exclamation) in the example below, is taken to be 1.

```
In [88]: df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})
In [89]: df;

>>> df = pd.DataFrame({'a': ['xxx', u'¡¡'], 'b': ['yyy', u'¡¡']})
>>> df
   a    b
0 xxx  yyy
1  ¡¡  ¡¡
```

Enabling `display.unicode.ambiguous_as_wide` makes pandas interpret these characters’ widths to be 2. (Note that this option will only be effective when `display.unicode.east_asian_width` is enabled.)

However, setting this option incorrectly for your terminal will cause these characters to be aligned incorrectly:

```
In [90]: pd.set_option('display.unicode.ambiguous_as_wide', True)
In [91]: df;

>>> pd.set_option('display.unicode.ambiguous_as_wide', True)
>>> df
   a    b
0 xxx  yyy
1  ¡¡  ¡¡
```

## 11.8 Table Schema Display

New in version 0.20.0.

`DataFrame` and `Series` will publish a Table Schema representation by default. False by default, this can be enabled globally with the `display.html.table_schema` option:

```
In [92]: pd.set_option('display.html.table_schema', True)
```

Only `'display.max_rows'` are serialized and published.



## INDEXING AND SELECTING DATA

The axis labeling information in pandas objects serves many purposes:

- Identifies data (i.e. provides *metadata*) using known indicators, important for analysis, visualization, and interactive console display.
- Enables automatic and explicit data alignment.
- Allows intuitive getting and setting of subsets of the data set.

In this section, we will focus on the final point: namely, how to slice, dice, and generally get and set subsets of pandas objects. The primary focus will be on Series and DataFrame as they have received more development attention in this area.

---

**Note:** The Python and NumPy indexing operators `[]` and attribute operator `.` provide quick and easy access to pandas data structures across a wide range of use cases. This makes interactive work intuitive, as there's little new to learn if you already know how to deal with Python dictionaries and NumPy arrays. However, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits. For production code, we recommended that you take advantage of the optimized pandas data access methods exposed in this chapter.

---

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See [Returning a View versus Copy](#).

**Warning:** Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see [here](#).

See the [MultiIndex / Advanced Indexing](#) for `MultiIndex` and more advanced indexing documentation.

See the [cookbook](#) for some advanced strategies.

### 12.1 Different Choices for Indexing

Object selection has had a number of user-requested additions in order to support more explicit location based indexing. Pandas now supports three types of multi-axis indexing.

- `.loc` is primarily label based, but may also be used with a boolean array. `.loc` will raise `KeyError` when the items are not found. Allowed inputs are:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a *label* of the index. This use is **not** an integer position along the index.).
- A list or array of labels ['a', 'b', 'c'].
- A slice object with labels 'a': 'f' (Note that contrary to usual python slices, **both** the start and the stop are included, when present in the index! See [Slicing with labels](#).).
- A boolean array
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above).

New in version 0.18.1.

See more at [Selection by Label](#).

- `.iloc` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a boolean array. `.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing. (this conforms with Python/NumPy *slice* semantics). Allowed inputs are:
  - An integer e.g. 5.
  - A list or array of integers [4, 3, 0].
  - A slice object with ints 1:7.
  - A boolean array.
  - A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above).

New in version 0.18.1.

See more at [Selection by Position](#), [Advanced Indexing](#) and [Advanced Hierarchical](#).

- `.loc`, `.iloc`, and also `[]` indexing can accept a callable as indexer. See more at [Selection By Callable](#).

Getting values from an object with multi-axes selection uses the following notation (using `.loc` as an example, but the following applies to `.iloc` as well). Any of the axes accessors may be the null slice `:`. Axes left out of the specification are assumed to be `:`, e.g. `p.loc['a']` is equivalent to `p.loc['a', :, :]`.

| Object Type | Indexers   |
|-------------|--|
| Series      | <code>s.loc[indexer]</code>                                    |
| DataFrame   | <code>df.loc[row_indexer, column_indexer]</code>               |
| Panel       | <code>p.loc[item_indexer, major_indexer, minor_indexer]</code> |

## 12.2 Basics

As mentioned when introducing the data structures in the [last section](#), the primary function of indexing with `[]` (a.k.a. `__getitem__` for those familiar with implementing class behavior in Python) is selecting out lower-dimensional slices. The following table shows return type values when indexing pandas objects with `[]`:

| Object Type | Selection                    | Return Value Type                       |
|-------------|------------------------------|---|
| Series      | <code>series[label]</code>   | scalar value                            |
| DataFrame   | <code>frame[colname]</code>  | Series corresponding to colname         |
| Panel       | <code>panel[itemname]</code> | DataFrame corresponding to the itemname |

Here we construct a simple time series data set to use for illustrating the indexing functionality:

```
In [1]: dates = pd.date_range('1/1/2000', periods=8)

In [2]: df = pd.DataFrame(np.random.randn(8, 4), index=dates, columns=['A', 'B', 'C',
↪ 'D'])

In [3]: df
Out[3]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | 0.469112  | -0.282863 | -1.509059 | -1.135632 |
| 2000-01-02 | 1.212112  | -0.173215 | 0.119209  | -1.044236 |
| 2000-01-03 | -0.861849 | -2.104569 | -0.494929 | 1.071804  |
| 2000-01-04 | 0.721555  | -0.706771 | -1.039575 | 0.271860  |
| 2000-01-05 | -0.424972 | 0.567020  | 0.276232  | -1.087401 |
| 2000-01-06 | -0.673690 | 0.113648  | -1.478427 | 0.524988  |
| 2000-01-07 | 0.404705  | 0.577046  | -1.715002 | -1.039268 |
| 2000-01-08 | -0.370647 | -1.157892 | -1.344312 | 0.844885  |

```
In [4]: panel = pd.Panel({'one' : df, 'two' : df - df.mean()})

In [5]: panel
Out[5]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 8 (major_axis) x 4 (minor_axis)
Items axis: one to two
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-08 00:00:00
Minor_axis axis: A to D
```

**Note:** None of the indexing functionality is time series specific unless specifically stated.

Thus, as per above, we have the most basic indexing using []:

```
In [6]: s = df['A']

In [7]: s[dates[5]]
Out[7]: -0.67368970808837059

In [8]: panel['two']
Out[8]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | 0.409571  | 0.113086  | -0.610826 | -0.936507 |
| 2000-01-02 | 1.152571  | 0.222735  | 1.017442  | -0.845111 |
| 2000-01-03 | -0.921390 | -1.708620 | 0.403304  | 1.270929  |
| 2000-01-04 | 0.662014  | -0.310822 | -0.141342 | 0.470985  |
| 2000-01-05 | -0.484513 | 0.962970  | 1.174465  | -0.888276 |
| 2000-01-06 | -0.733231 | 0.509598  | -0.580194 | 0.724113  |
| 2000-01-07 | 0.345164  | 0.972995  | -0.816769 | -0.840143 |
| 2000-01-08 | -0.430188 | -0.761943 | -0.446079 | 1.044010  |

You can pass a list of columns to [] to select columns in that order. If a column is not contained in the DataFrame, an exception will be raised. Multiple columns can also be set in this manner:

```
In [9]: df
Out[9]:
```

|  | A | B | C | D |
|--|---|---|---|---|
|--|---|---|---|---|

(continues on next page)

(continued from previous page)

```

2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885

```

```
In [10]: df[['B', 'A']] = df[['A', 'B']]
```

```
In [11]: df
```

```
Out[11]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | -0.282863 | 0.469112  | -1.509059 | -1.135632 |
| 2000-01-02 | -0.173215 | 1.212112  | 0.119209  | -1.044236 |
| 2000-01-03 | -2.104569 | -0.861849 | -0.494929 | 1.071804  |
| 2000-01-04 | -0.706771 | 0.721555  | -1.039575 | 0.271860  |
| 2000-01-05 | 0.567020  | -0.424972 | 0.276232  | -1.087401 |
| 2000-01-06 | 0.113648  | -0.673690 | -1.478427 | 0.524988  |
| 2000-01-07 | 0.577046  | 0.404705  | -1.715002 | -1.039268 |
| 2000-01-08 | -1.157892 | -0.370647 | -1.344312 | 0.844885  |

You may find this useful for applying a transform (in-place) to a subset of the columns.

**Warning:** pandas aligns all AXES when setting Series and DataFrame from `.loc`, and `.iloc`.

This will **not** modify `df` because the column alignment is before value assignment.

```
In [12]: df[['A', 'B']]
```

```
Out[12]:
```

|            | A         | B         |
|------------|-----------|-----------|
| 2000-01-01 | -0.282863 | 0.469112  |
| 2000-01-02 | -0.173215 | 1.212112  |
| 2000-01-03 | -2.104569 | -0.861849 |
| 2000-01-04 | -0.706771 | 0.721555  |
| 2000-01-05 | 0.567020  | -0.424972 |
| 2000-01-06 | 0.113648  | -0.673690 |
| 2000-01-07 | 0.577046  | 0.404705  |
| 2000-01-08 | -1.157892 | -0.370647 |

```
In [13]: df.loc[:, ['B', 'A']] = df[['A', 'B']]
```

```
In [14]: df[['A', 'B']]
```

```
Out[14]:
```

|            | A         | B         |
|------------|-----------|-----------|
| 2000-01-01 | -0.282863 | 0.469112  |
| 2000-01-02 | -0.173215 | 1.212112  |
| 2000-01-03 | -2.104569 | -0.861849 |
| 2000-01-04 | -0.706771 | 0.721555  |
| 2000-01-05 | 0.567020  | -0.424972 |
| 2000-01-06 | 0.113648  | -0.673690 |
| 2000-01-07 | 0.577046  | 0.404705  |
| 2000-01-08 | -1.157892 | -0.370647 |

The correct way to swap column values is by using raw values:

You may access an index on a `Series`, column on a `DataFrame`, and an item on a `Panel` directly as an attribute:

```
In [18]: dfa = df.copy()
```

Out [19]: 2

```
\\\\\\\\\\\\\\\\\\\\Out[20]:
```

```
In [21]: panel.one
```

```
In [22]: sa.a = 5
```

```
In [23]: sa
```

---

(continues on next page)

(continued from previous page)

```

Out [23]:
a      5
b      2
c      3
dtype: int64

In [24]: dfa.A = list(range(len(dfa.index))) # ok if A already exists

In [25]: dfa
Out [25]:
           A          B          C          D
2000-01-01  0 -0.282863 -1.509059 -1.135632
2000-01-02  1 -0.173215  0.119209 -1.044236
2000-01-03  2 -2.104569 -0.494929  1.071804
2000-01-04  3 -0.706771 -1.039575  0.271860
2000-01-05  4  0.567020  0.276232 -1.087401
2000-01-06  5  0.113648 -1.478427  0.524988
2000-01-07  6  0.577046 -1.715002 -1.039268
2000-01-08  7 -1.157892 -1.344312  0.844885

In [26]: dfa['A'] = list(range(len(dfa.index))) # use this form to create a new_
↪column

In [27]: dfa
Out [27]:
           A          B          C          D
2000-01-01  0 -0.282863 -1.509059 -1.135632
2000-01-02  1 -0.173215  0.119209 -1.044236
2000-01-03  2 -2.104569 -0.494929  1.071804
2000-01-04  3 -0.706771 -1.039575  0.271860
2000-01-05  4  0.567020  0.276232 -1.087401
2000-01-06  5  0.113648 -1.478427  0.524988
2000-01-07  6  0.577046 -1.715002 -1.039268
2000-01-08  7 -1.157892 -1.344312  0.844885

```

**Warning:**

- You can use this access only if the index element is a valid Python identifier, e.g. `s.1` is not allowed. See [here for an explanation of valid identifiers](#).
- The attribute will not be available if it conflicts with an existing method name, e.g. `s.min` is not allowed.
- Similarly, the attribute will not be available if it conflicts with any of the following list: `index`, `major_axis`, `minor_axis`, `items`.
- In any of these cases, standard indexing will still work, e.g. `s['1']`, `s['min']`, and `s['index']` will access the corresponding element or column.

If you are using the IPython environment, you may also use tab-completion to see these accessible attributes.

You can also assign a `dict` to a row of a `DataFrame`:

```

In [28]: x = pd.DataFrame({'x': [1, 2, 3], 'y': [3, 4, 5]})

In [29]: x.iloc[1] = dict(x=9, y=99)

```

(continues on next page)

(continued from previous page)

```
In [30]: x
```

Out [30] :

|   | x | y  |
|---|---|----|
| 0 | 1 | 3  |
| 1 | 9 | 99 |
| 2 | 3 | 5  |

You can use attribute access to modify an existing element of a Series or column of a DataFrame, but be careful; if you try to use attribute access to create a new column, it creates a new attribute rather than a new column. In 0.21.0 and later, this will raise a `UserWarning`:

```
In[1]: df = pd.DataFrame({'one': [1., 2., 3.]})
```

```
In[2]: df.two = [4, 5, 6]
```

```
UserWarning: Pandas doesn't allow Series to be assigned into nonexistent columns -
↳ see https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute\_access
```

```
In[3]: df
```

Out [3]:

|   | one |
|---|-----|
| 0 | 1.0 |
| 1 | 2.0 |
| 2 | 3.0 |

## 12.4 Slicing ranges

The most robust and consistent way of slicing ranges along arbitrary axes is described in the *Selection by Position* section detailing the `.iloc` method. For now, we explain the semantics of slicing using the `[]` operator.

With `Series`, the syntax works exactly as with an `ndarray`, returning a slice of the values and the corresponding labels:

```
In [31]: s[:5]
```

Out [31]:

```
2000-01-01    0.469112
2000-01-02    1.212112
2000-01-03   -0.861849
2000-01-04    0.721555
2000-01-05   -0.424972
Freq: D, Name: A, dtype: float64
```

```
In [32]: s[::2]
```

```

2000-01-01    0.469112
2000-01-03   -0.861849
2000-01-05   -0.424972
2000-01-07    0.404705
Freq: 2D, Name: A, dtype: float64

```

```
In [33]: s[::-1]
```

↪

|            |           |
|------------|-----------|
| 2000-01-08 | -0.370647 |
| 2000-01-07 | 0.404705  |
| 2000-01-06 | -0.673690 |
| 2000-01-05 | -0.424972 |

(continues on next page)

(continued from previous page)

```

2000-01-04    0.721555
2000-01-03   -0.861849
2000-01-02    1.212112
2000-01-01    0.469112
Freq: -1D, Name: A, dtype: float64

```

Note that setting works as well:

```

In [34]: s2 = s.copy()

In [35]: s2[:5] = 0

In [36]: s2
Out[36]:
2000-01-01    0.000000
2000-01-02    0.000000
2000-01-03    0.000000
2000-01-04    0.000000
2000-01-05    0.000000
2000-01-06   -0.673690
2000-01-07    0.404705
2000-01-08   -0.370647
Freq: D, Name: A, dtype: float64

```

With DataFrame, slicing inside of `[]` **slices the rows**. This is provided largely as a convenience since it is such a common operation.

```

In [37]: df[:3]
Out[37]:
           A          B          C          D
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804

In [38]: df[::-1]
Out[38]:
           A          B          C          D
2000-01-08 -0.370647 -1.157892 -1.344312  0.844885
2000-01-07  0.404705  0.577046 -1.715002 -1.039268
2000-01-06 -0.673690  0.113648 -1.478427  0.524988
2000-01-05 -0.424972  0.567020  0.276232 -1.087401
2000-01-04  0.721555 -0.706771 -1.039575  0.271860
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804
2000-01-02  1.212112 -0.173215  0.119209 -1.044236
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632

```

## 12.5 Selection By Label

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See *Returning a View versus Copy*.



**Warning:**

.loc is strict when you present slicers that are not compatible (or convertible) with the index type. For example using integers in a DatetimeIndex. These will raise a TypeError.

```
In [39]: df1 = pd.DataFrame(np.random.randn(5,4), columns=list('ABCD'), index=pd.
↳date_range('20130101',periods=5))
```

```
In [40]: df1
```

```
Out [40]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-01 | 1.075770  | -0.109050 | 1.643563  | -1.469388 |
| 2013-01-02 | 0.357021  | -0.674600 | -1.776904 | -0.968914 |
| 2013-01-03 | -1.294524 | 0.413738  | 0.276662  | -0.472035 |
| 2013-01-04 | -0.013960 | -0.362543 | -0.006154 | -0.923061 |
| 2013-01-05 | 0.895717  | 0.805244  | -1.206412 | 2.565646  |

```
In [4]: df1.loc[2:3]
```

```
TypeError: cannot do slice indexing on <class 'pandas.tseries.index.DatetimeIndex'>
↳with these indexers [2] of <type 'int'>
```

String likes in slicing *can* be convertible to the type of the index and lead to natural slicing.

```
In [41]: df1.loc['20130102':'20130104']
```

```
Out [41]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2013-01-02 | 0.357021  | -0.674600 | -1.776904 | -0.968914 |
| 2013-01-03 | -1.294524 | 0.413738  | 0.276662  | -0.472035 |
| 2013-01-04 | -0.013960 | -0.362543 | -0.006154 | -0.923061 |

**Warning:** Starting in 0.21.0, pandas will show a FutureWarning if indexing with a list with missing labels. In the future this will raise a KeyError. See *list-like Using loc with missing keys in a list is Deprecated*.

pandas provides a suite of methods in order to have **purely label based indexing**. This is a strict inclusion based protocol. Every label asked for must be in the index, or a `KeyError` will be raised. When slicing, both the start bound **AND** the stop bound are *included*, if present in the index. Integers are valid labels, but they refer to the label **and not the position**.

The `.loc` attribute is the primary access method. The following are valid inputs:

- A single label, e.g. 5 or 'a' (Note that 5 is interpreted as a *label* of the index. This use is **not** an integer position along the index.).
- A list or array of labels ['a', 'b', 'c'].
- A slice object with labels 'a': 'f' (Note that contrary to usual python slices, **both** the start and the stop are included, when present in the index! See *Slicing with labels*.).
- A boolean array.
- A callable, see *Selection By Callable*.

```
In [42]: s1 = pd.Series(np.random.randn(6), index=list('abcdef'))
```

```
In [43]: s1
```

```
Out [43]:
```

|   |          |
|---|----------|
| a | 1.431256 |
|---|----------|

(continues on next page)

(continued from previous page)

```
b    1.340309
c   -1.170299
d   -0.226169
e    0.410835
f    0.813850
dtype: float64
```

```
In [44]: s1.loc['c':]
```

```

////////////////////////////////////
↪
c   -1.170299
d   -0.226169
e    0.410835
f    0.813850
dtype: float64
```

```
In [45]: s1.loc['b']
```

```

////////////////////////////////////
↪1.3403088497993827
```

Note that setting works as well:

```
In [46]: s1.loc['c':] = 0
```

```
In [47]: s1
```

```
Out[47]:
```

```
a    1.431256
b    1.340309
c    0.000000
d    0.000000
e    0.000000
f    0.000000
dtype: float64
```

With a DataFrame:

```
In [48]: df1 = pd.DataFrame(np.random.randn(6,4),
.....:                      index=list('abcdef'),
.....:                      columns=list('ABCD'))
.....:
```

```
In [49]: df1
```

```
Out[49]:
```

```

      A         B         C         D
a  0.132003 -0.827317 -0.076467 -1.187678
b  1.130127 -1.436737 -1.413681  1.607920
c  1.024180  0.569605  0.875906 -2.211372
d  0.974466 -2.006747 -0.410001 -0.078638
e  0.545952 -1.219217 -1.226825  0.769804
f -1.281247 -0.727707 -0.121306 -0.097883
```

```
In [50]: df1.loc[['a', 'b', 'd'], :]
```

```

////////////////////////////////////
↪
      A         B         C         D
a  0.132003 -0.827317 -0.076467 -1.187678
b  1.130127 -1.436737 -1.413681  1.607920
```

(continues on next page)

(continued from previous page)

```
d  0.974466 -2.006747 -0.410001 -0.078638
```

Accessing via label slices:

```
In [51]: df1.loc['d':, 'A':'C']
Out[51]:
```

|   | A         | B         | C         |
|---|-----------|-----------|-----------|
| d | 0.974466  | -2.006747 | -0.410001 |
| e | 0.545952  | -1.219217 | -1.226825 |
| f | -1.281247 | -0.727707 | -0.121306 |

For getting a cross section using a label (equivalent to `df.xs('a')`):

```
In [52]: df1.loc['a']
Out[52]:
```

|   | A         |
|---|-----------|
| A | 0.132003  |
| B | -0.827317 |
| C | -0.076467 |
| D | -1.187678 |

Name: a, dtype: float64

For getting values with a boolean array:

```
In [53]: df1.loc['a'] > 0
Out[53]:
```

|   | A     |
|---|-------|
| A | True  |
| B | False |
| C | False |
| D | False |

Name: a, dtype: bool

```
In [54]: df1.loc[:, df1.loc['a'] > 0]
Out[54]:
```

|   | A         |
|---|-----------|
| a | 0.132003  |
| b | 1.130127  |
| c | 1.024180  |
| d | 0.974466  |
| e | 0.545952  |
| f | -1.281247 |

For getting a value explicitly (equivalent to deprecated `df.get_value('a', 'A')`):

```
# this is also equivalent to ``df1.at['a', 'A']``
In [55]: df1.loc['a', 'A']
Out[55]: 0.13200317033032932
```

### 12.5.1 Slicing with labels

When using `.loc` with slices, if both the start and the stop labels are present in the index, then elements *located* between the two (including them) are returned:

```
In [56]: s = pd.Series(list('abcde'), index=[0,3,2,5,4])
In [57]: s.loc[3:5]
```

(continues on next page)

(continued from previous page)

```
Out [57]:
3      b
2      c
5      d
dtype: object
```

If at least one of the two is absent, but the index is sorted, and can be compared against start and stop labels, then slicing will still work as expected, by selecting labels which *rank* between the two:

```
In [58]: s.sort_index()
Out [58]:
0      a
2      c
3      b
4      e
5      d
dtype: object

In [59]: s.sort_index().loc[1:6]
Out [59]:
2      c
3      b
4      e
5      d
dtype: object
```

However, if at least one of the two is absent *and* the index is not sorted, an error will be raised (since doing otherwise would be computationally expensive, as well as potentially ambiguous for mixed type indexes). For instance, in the above example, `s.loc[1:6]` would raise `KeyError`.

## 12.6 Selection By Position

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See [Returning a View versus Copy](#).

Pandas provides a suite of methods in order to get **purely integer based indexing**. The semantics follow closely Python and NumPy slicing. These are 0-based indexing. When slicing, the start bounds is *included*, while the upper bound is *excluded*. Trying to use a non-integer, even a **valid** label will raise an `IndexError`.

The `.iloc` attribute is the primary access method. The following are valid inputs:

- An integer e.g. 5.
- A list or array of integers `[4, 3, 0]`.
- A slice object with ints `1:7`.
- A boolean array.
- A callable, see [Selection By Callable](#).

```
In [60]: s1 = pd.Series(np.random.randn(5), index=list(range(0,10,2)))
In [61]: s1
```

(continues on next page)

(continued from previous page)

**Out [61]:**

```
0    0.695775
2    0.341734
4    0.959726
6   -1.110336
8   -0.619976
dtype: float64
```

**In [62]:** s1.iloc[:3]

```
Out [62]:
0    0.695775
2    0.341734
4    0.959726
dtype: float64
```

**In [63]:** s1.iloc[3]

```
Out [63]:
-1.1103361028911669
```

Note that setting works as well:

**In [64]:** s1.iloc[:3] = 0**In [65]:** s1**Out [65]:**

```
0    0.000000
2    0.000000
4    0.000000
6   -1.110336
8   -0.619976
dtype: float64
```

With a DataFrame:

```
In [66]: df1 = pd.DataFrame(np.random.randn(6,4),
.....:                      index=list(range(0,12,2)),
.....:                      columns=list(range(0,8,2)))
.....:
```

**In [67]:** df1**Out [67]:**

```
      0         2         4         6
0  0.149748 -0.732339  0.687738  0.176444
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161
6 -0.826591 -0.345352  1.314232  0.690579
8  0.995761  2.396780  0.014871  3.357427
10 -0.317441 -1.236269  0.896171 -0.487602
```

Select via integer slicing:

**In [68]:** df1.iloc[:3]**Out [68]:**

```
      0         2         4         6
0  0.149748 -0.732339  0.687738  0.176444
2  0.403310 -0.154951  0.301624 -2.179861
```

(continues on next page)

(continued from previous page)

```
4 -1.369849 -0.954208 1.462696 -1.743161
```

```
In [69]: df1.iloc[1:5, 2:4]
```

```
////////////////////////////////////
```

```
↪
      4      6
2  0.301624 -2.179861
4  1.462696 -1.743161
6  1.314232  0.690579
8  0.014871  3.357427
```

Select via integer list:

```
In [70]: df1.iloc[[1, 3, 5], [1, 3]]
```

```
Out [70]:
```

```
      2      6
2 -0.154951 -2.179861
6 -0.345352  0.690579
10 -1.236269 -0.487602
```

```
In [71]: df1.iloc[1:3, :]
```

```
Out [71]:
```

```
      0      2      4      6
2  0.403310 -0.154951  0.301624 -2.179861
4 -1.369849 -0.954208  1.462696 -1.743161
```

```
In [72]: df1.iloc[:, 1:3]
```

```
Out [72]:
```

```
      2      4
0 -0.732339  0.687738
2 -0.154951  0.301624
4 -0.954208  1.462696
6 -0.345352  1.314232
8  2.396780  0.014871
10 -1.236269  0.896171
```

```
# this is also equivalent to ``df1.iat[1,1]``
```

```
In [73]: df1.iloc[1, 1]
```

```
Out [73]: -0.15495077442490321
```

For getting a cross section using an integer position (equiv to `df.xs(1)`):

```
In [74]: df1.iloc[1]
```

```
Out [74]:
```

```
0    0.403310
2   -0.154951
4    0.301624
6   -2.179861
Name: 2, dtype: float64
```

Out of range slice indexes are handled gracefully just as in Python/Numpy.

```
# these are allowed in python/numpy.
```

```
In [75]: x = list('abcdef')
```

(continues on next page)

(continued from previous page)

```

In [76]: x
Out[76]: ['a', 'b', 'c', 'd', 'e', 'f']

In [77]: x[4:10]
\\Out[77]: ['e', 'f']

In [78]: x[8:10]
\\Out[78]: []

In [79]: s = pd.Series(x)

In [80]: s
Out[80]:
0    a
1    b
2    c
3    d
4    e
5    f
dtype: object

In [81]: s.iloc[4:10]
\\Out[81]:
4    e
5    f
dtype: object

In [82]: s.iloc[8:10]
\\Out[82]:
↪Series([], dtype: object)

```

Note that using slices that go out of bounds can result in an empty axis (e.g. an empty DataFrame being returned).

```

In [83]: df1 = pd.DataFrame(np.random.randn(5,2), columns=list('AB'))

In [84]: df1
Out[84]:
      A      B
0 -0.082240 -2.182937
1  0.380396  0.084844
2  0.432390  1.519970
3 -0.493662  0.600178
4  0.274230  0.132885

In [85]: df1.iloc[:, 2:3]
↪
Empty DataFrame
Columns: []
Index: [0, 1, 2, 3, 4]

In [86]: df1.iloc[:, 1:3]
↪
      B
0 -2.182937
1  0.084844

```

(continues on next page)

(continued from previous page)

```
2  1.519970
3  0.600178
4  0.132885
```

```
In [87]: df1.iloc[4:6]
```

```

////////////////////////////////////
↪
      A      B
4  0.27423  0.132885
```

A single indexer that is out of bounds will raise an `IndexError`. A list of indexers where any element is out of bounds will raise an `IndexError`.

```
df1.iloc[[4, 5, 6]]
IndexError: positional indexers are out-of-bounds

df1.iloc[:, 4]
IndexError: single positional indexer is out-of-bounds
```

## 12.7 Selection By Callable

New in version 0.18.1.

`.loc`, `.iloc`, and also `[]` indexing can accept a callable as indexer. The callable must be a function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing.

```
In [88]: df1 = pd.DataFrame(np.random.randn(6, 4),
.....:                      index=list('abcdef'),
.....:                      columns=list('ABCD'))
.....:
```

```
In [89]: df1
```

```
Out[89]:
      A      B      C      D
a -0.023688  2.410179  1.450520  0.206053
b -0.251905 -2.213588  1.063327  1.266143
c  0.299368 -0.863838  0.408204 -1.048089
d -0.025747 -0.988387  0.094055  1.262731
e  1.289997  0.082423 -0.055758  0.536580
f -0.489682  0.369374 -0.034571 -2.484478
```

```
In [90]: df1.loc[lambda df: df.A > 0, :]
```

```

////////////////////////////////////
↪
      A      B      C      D
c  0.299368 -0.863838  0.408204 -1.048089
e  1.289997  0.082423 -0.055758  0.536580
```

```
In [91]: df1.loc[:, lambda df: ['A', 'B']]
```

```

////////////////////////////////////
↪
      A      B
a -0.023688  2.410179
b -0.251905 -2.213588
```

(continues on next page)



(continued from previous page)

```
c 0.299368 -0.863838
d -0.025747 -0.988387
e 1.289997 0.082423
f -0.489682 0.369374
```

```
In [92]: df1.iloc[:, lambda df: [0, 1]]
```

```
=====
↪
      A      B
a -0.023688  2.410179
b -0.251905 -2.213588
c  0.299368 -0.863838
d -0.025747 -0.988387
e  1.289997  0.082423
f -0.489682  0.369374
```

```
In [93]: df1[lambda df: df.columns[0]]
```

```
=====
↪
a -0.023688
b -0.251905
c  0.299368
d -0.025747
e  1.289997
f -0.489682
Name: A, dtype: float64
```

You can use callable indexing in Series.

```
In [94]: df1.A.loc[lambda s: s > 0]
```

```
Out [94]:
c    0.299368
e    1.289997
Name: A, dtype: float64
```

Using these methods / indexers, you can chain data selection operations without using temporary variable.

```
In [95]: bb = pd.read_csv('data/baseball.csv', index_col='id')
```

```
In [96]: (bb.groupby(['year', 'team']).sum()
....:      .loc[lambda df: df.r > 100])
....:
```

```
Out [96]:
      stint  g  ab  r  h  X2b  X3b  hr  rbi  sb  cs  bb  so
↪ibb  hbp  sh  sf  gidp
year team
↪
2007 CIN      6  379   745  101  203   35    2  36  125.0  10.0  1.0  105  127.0  14.
↪0    1.0   1.0  15.0  18.0
    DET      5  301  1062  162  283   54    4  37  144.0  24.0  7.0   97  176.0   3.
↪0   10.0   4.0   8.0  28.0
    HOU      4  311   926  109  218   47    6  14   77.0  10.0  4.0   60  212.0   3.
↪0    9.0  16.0   6.0  17.0
    LAN     11  413  1021  153  293   61    3  36  154.0   7.0  5.0  114  141.0   8.
↪0    9.0   3.0   8.0  29.0
    NYN     13  622  1854  240  509  101    3  61  243.0  22.0  4.0  174  310.0  24.
↪0   23.0  18.0  15.0  48.0
```

(continues on next page)

(continued from previous page)

|    |      |      |      |      |     |     |    |   |    |       |      |     |     |       |     |
|----|------|------|------|------|-----|-----|----|---|----|-------|------|-----|-----|-------|-----|
|    | SFN  | 5    | 482  | 1305 | 198 | 337 | 67 | 6 | 40 | 171.0 | 26.0 | 7.0 | 235 | 188.0 | 51. |
| ↪0 | 8.0  | 16.0 | 6.0  | 41.0 |     |     |    |   |    |       |      |     |     |       |     |
|    | TEX  | 2    | 198  | 729  | 115 | 200 | 40 | 4 | 28 | 115.0 | 21.0 | 4.0 | 73  | 140.0 | 4.  |
| ↪0 | 5.0  | 2.0  | 8.0  | 16.0 |     |     |    |   |    |       |      |     |     |       |     |
|    | TOR  | 4    | 459  | 1408 | 187 | 378 | 96 | 2 | 58 | 223.0 | 4.0  | 2.0 | 190 | 265.0 | 16. |
| ↪0 | 12.0 | 4.0  | 16.0 | 38.0 |     |     |    |   |    |       |      |     |     |       |     |

## 12.8 IX Indexer is Deprecated

**Warning:** Starting in 0.20.0, the `.ix` indexer is deprecated, in favor of the more strict `.iloc` and `.loc` indexers.

`.ix` offers a lot of magic on the inference of what the user wants to do. To wit, `.ix` can decide to index *positionally* OR via *labels* depending on the data type of the index. This has caused quite a bit of user confusion over the years.

The recommended methods of indexing are:

- `.loc` if you want to *label* index.
- `.iloc` if you want to *positionally* index.

```
In [97]: dfd = pd.DataFrame({'A': [1, 2, 3],
.....:                      'B': [4, 5, 6]},
.....:                      index=list('abc'))
.....:
```

```
In [98]: dfd
```

```
Out[98]:
```

```
   A  B
a  1  4
b  2  5
c  3  6
```

Previous behavior, where you wish to get the 0th and the 2nd elements from the index in the 'A' column.

```
In [3]: dfd.ix[[0, 2], 'A']
```

```
Out[3]:
```

```
a    1
c    3
Name: A, dtype: int64
```

Using `.loc`. Here we will select the appropriate indexes from the index, then use *label* indexing.

```
In [99]: dfd.loc[dfd.index[[0, 2]], 'A']
```

```
Out[99]:
```

```
a    1
c    3
Name: A, dtype: int64
```

This can also be expressed using `.iloc`, by explicitly getting locations on the indexers, and using *positional* indexing to select things.

```
In [100]: dfd.iloc[[0, 2], dfd.columns.get_loc('A')]
```

```
Out[100]:
```

(continues on next page)

(continued from previous page)

```
a    1
c    3
Name: A, dtype: int64
```

For getting *multiple* indexers, using `.get_indexer`:

```
In [101]: dfd.iloc[[0, 2], dfd.columns.get_indexer(['A', 'B'])]
Out[101]:
   A  B
a  1  4
c  3  6
```

## 12.9 Indexing with list with missing labels is Deprecated

**Warning:** Starting in 0.21.0, using `.loc` or `[]` with a list with one or more missing labels, is deprecated, in favor of `.reindex`.

In prior versions, using `.loc[list-of-labels]` would work as long as *at least 1* of the keys was found (otherwise it would raise a `KeyError`). This behavior is deprecated and will show a warning message pointing to this section. The recommended alternative is to use `.reindex()`.

For example.

```
In [102]: s = pd.Series([1, 2, 3])

In [103]: s
Out[103]:
0    1
1    2
2    3
dtype: int64
```

Selection with all keys found is unchanged.

```
In [104]: s.loc[[1, 2]]
Out[104]:
1    2
2    3
dtype: int64
```

Previous Behavior

```
In [4]: s.loc[[1, 2, 3]]
Out[4]:
1    2.0
2    3.0
3    NaN
dtype: float64
```

Current Behavior

```
In [4]: s.loc[[1, 2, 3]]
Passing list-likes to .loc with any non-matching elements will raise
KeyError in the future, you can use .reindex() as an alternative.

See the documentation here:
http://pandas.pydata.org/pandas-docs/stable/indexing.html#deprecate-loc-reindex-
→listlike

Out[4]:
1      2.0
2      3.0
3      NaN
dtype: float64
```

## 12.9.1 Reindexing

The idiomatic way to achieve selecting potentially not-found elements is via `.reindex()`. See also the section on *reindexing*.

```
In [105]: s.reindex([1, 2, 3])
Out[105]:
1      2.0
2      3.0
3      NaN
dtype: float64
```

Alternatively, if you want to select only *valid* keys, the following is idiomatic and efficient; it is guaranteed to preserve the dtype of the selection.

```
In [106]: labels = [1, 2, 3]

In [107]: s.loc[s.index.intersection(labels)]
Out[107]:
1      2
2      3
dtype: int64
```

Having a duplicated index will raise for a `.reindex()`:

```
In [108]: s = pd.Series(np.arange(4), index=['a', 'a', 'b', 'c'])

In [109]: labels = ['c', 'd']
```

```
In [17]: s.reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

Generally, you can intersect the desired labels with the current axis, and then reindex.

```
In [110]: s.loc[s.index.intersection(labels)].reindex(labels)
Out[110]:
c      3.0
d      NaN
dtype: float64
```

However, this would *still* raise if your resulting index is duplicated.

```
In [41]: labels = ['a', 'd']

In [42]: s.loc[s.index.intersection(labels)].reindex(labels)
ValueError: cannot reindex from a duplicate axis
```

## 12.10 Selecting Random Samples

A random selection of rows or columns from a Series, DataFrame, or Panel with the `sample()` method. The method will sample rows by default, and accepts a specific number of rows/columns to return, or a fraction of rows.

```
In [111]: s = pd.Series([0,1,2,3,4,5])

# When no arguments are passed, returns 1 row.
In [112]: s.sample()
Out[112]:
4      4
dtype: int64

# One may specify either a number of rows:
In [113]: s.sample(n=3)
Out[113]:
0      0
4      4
1      1
dtype: int64

# Or a fraction of the rows:
In [114]: s.sample(frac=0.5)
Out[114]:
5      5
3      3
1      1
dtype: int64
```

By default, `sample` will return each row at most once, but one can also sample with replacement using the `replace` option:

```
In [115]: s = pd.Series([0,1,2,3,4,5])

# Without replacement (default):
In [116]: s.sample(n=6, replace=False)
Out[116]:
0      0
1      1
5      5
3      3
2      2
4      4
dtype: int64

# With replacement:
In [117]: s.sample(n=6, replace=True)
Out[117]:
0      0
4      4
```

(continues on next page)

(continued from previous page)

```
3      3
2      2
4      4
4      4
dtype: int64
```

By default, each row has an equal probability of being selected, but if you want rows to have different probabilities, you can pass the `sample` function sampling weights as `weights`. These weights can be a list, a NumPy array, or a Series, but they must be of the same length as the object you are sampling. Missing values will be treated as a weight of zero, and `inf` values are not allowed. If weights do not sum to 1, they will be re-normalized by dividing all weights by the sum of the weights. For example:

```
In [118]: s = pd.Series([0,1,2,3,4,5])

In [119]: example_weights = [0, 0, 0.2, 0.2, 0.2, 0.4]

In [120]: s.sample(n=3, weights=example_weights)
Out[120]:
5      5
4      4
3      3
dtype: int64

# Weights will be re-normalized automatically
In [121]: example_weights2 = [0.5, 0, 0, 0, 0, 0]

In [122]: s.sample(n=1, weights=example_weights2)
Out[122]:
0      0
dtype: int64
```

When applied to a DataFrame, you can use a column of the DataFrame as sampling weights (provided you are sampling rows and not columns) by simply passing the name of the column as a string.

```
In [123]: df2 = pd.DataFrame({'col1':[9,8,7,6], 'weight_column':[0.5, 0.4, 0.1, 0]})

In [124]: df2.sample(n = 3, weights = 'weight_column')
Out[124]:
   col1  weight_column
1      8             0.4
0      9             0.5
2      7             0.1
```

`sample` also allows users to sample columns instead of rows using the `axis` argument.

```
In [125]: df3 = pd.DataFrame({'col1':[1,2,3], 'col2':[2,3,4]})

In [126]: df3.sample(n=1, axis=1)
Out[126]:
   col1
0      1
1      2
2      3
```

Finally, one can also set a seed for `sample`'s random number generator using the `random_state` argument, which will accept either an integer (as a seed) or a NumPy RandomState object.

## 12.11 Setting With Enlargement

The `.loc/[]` operations can perform enlargement when setting a non-existent key for that axis.

In the `Series` case this is effectively an appending operation.

```
In [130]: se = pd.Series([1,2,3])
```

```
In [131]: se
```

Out [131] :

```
0      1
1      2
2      3
dtype: int64
```

```
In [132]: se[5] = 5.
```

In [133]: se

```
Out[133]:
0      1.0
1      2.0
2      3.0
5      5.0
dtype: float64
```

A DataFrame can be enlarged on either axis via `.loc`.

```
In [134]: dfi = pd.DataFrame(np.arange(6).reshape(3,2),
.....:                        columns=['A', 'B'])
.....:
```

```
In [135]: dfi
```

Out [135] :

|   | A | B |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 2 | 3 |
| 2 | 4 | 5 |

```
In [136]: dfi.loc[:, 'C'] = dfi.loc[:, 'A']
```

(continues on next page)

(continued from previous page)

```
In [137]: dfi
Out[137]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
```

This is like an append operation on the DataFrame.

```
In [138]: dfi.loc[3] = 5

In [139]: dfi
Out[139]:
   A  B  C
0  0  1  0
1  2  3  2
2  4  5  4
3  5  5  5
```

## 12.12 Fast scalar value getting and setting

Since indexing with `[]` must handle a lot of cases (single-label access, slicing, boolean indexing, etc.), it has a bit of overhead in order to figure out what you're asking for. If you only want to access a scalar value, the fastest way is to use the `at` and `iat` methods, which are implemented on all of the data structures.

Similarly to `loc`, `at` provides **label** based scalar lookups, while, `iat` provides **integer** based lookups analogously to `iloc`

```
In [140]: s.iat[5]
Out[140]: 5

In [141]: df.at[dates[5], 'A']
Out[141]: -0.67368970808837059

In [142]: df.iat[3, 0]
Out[142]: 0.72155516224436689
```

You can also set using these same indexers.

```
In [143]: df.at[dates[5], 'E'] = 7

In [144]: df.iat[3, 0] = 7
```

`at` may enlarge the object in-place as above if the indexer is missing.

```
In [145]: df.at[dates[-1]+1, 0] = 7

In [146]: df
Out[146]:
              A              B              C              D              E              0
2000-01-01  0.469112 -0.282863 -1.509059 -1.135632      NaN      NaN
2000-01-02  1.212112 -0.173215  0.119209 -1.044236      NaN      NaN
2000-01-03 -0.861849 -2.104569 -0.494929  1.071804      NaN      NaN
2000-01-04  7.000000 -0.706771 -1.039575  0.271860      NaN      NaN
```

(continues on next page)



(continued from previous page)

|            |           |           |           |           |     |     |
|------------|-----------|-----------|-----------|-----------|-----|-----|
| 2000-01-05 | -0.424972 | 0.567020  | 0.276232  | -1.087401 | NaN | NaN |
| 2000-01-06 | -0.673690 | 0.113648  | -1.478427 | 0.524988  | 7.0 | NaN |
| 2000-01-07 | 0.404705  | 0.577046  | -1.715002 | -1.039268 | NaN | NaN |
| 2000-01-08 | -0.370647 | -1.157892 | -1.344312 | 0.844885  | NaN | NaN |
| 2000-01-09 | NaN       | NaN       | NaN       | NaN       | NaN | 7.0 |

## 12.13 Boolean indexing

Another common operation is the use of boolean vectors to filter the data. The operators are: `|` for `or`, `&` for `and`, and `~` for `not`. These **must** be grouped by using parentheses, since by default Python will evaluate an expression such as `df.A > 2 & df.B < 3` as `df.A > (2 & df.B) < 3`, while the desired evaluation order is `(df.A > 2) & (df.B < 3)`.

Using a boolean vector to index a Series works exactly as in a NumPy ndarray:

```
In [147]: s = pd.Series(range(-3, 4))

In [148]: s
Out[148]:
0    -3
1    -2
2    -1
3     0
4     1
5     2
6     3
dtype: int64

In [149]: s[s > 0]
Out[149]:
4     1
5     2
6     3
dtype: int64

In [150]: s[(s < -1) | (s > 0.5)]
Out[150]:
0    -3
1    -2
4     1
5     2
6     3
dtype: int64

In [151]: s[~(s < 0)]
Out[151]:
3     0
4     1
5     2
6     3
dtype: int64
```

You may select rows from a DataFrame using a boolean vector the same length as the DataFrame's index (for example,

something derived from one of the columns of the DataFrame):

```
In [152]: df[df['A'] > 0]
Out[152]:
```

|            | A        | B         | C         | D         | E   | 0   |
|------------|----------|-----------|-----------|-----------|-----|-----|
| 2000-01-01 | 0.469112 | -0.282863 | -1.509059 | -1.135632 | NaN | NaN |
| 2000-01-02 | 1.212112 | -0.173215 | 0.119209  | -1.044236 | NaN | NaN |
| 2000-01-04 | 7.000000 | -0.706771 | -1.039575 | 0.271860  | NaN | NaN |
| 2000-01-07 | 0.404705 | 0.577046  | -1.715002 | -1.039268 | NaN | NaN |

List comprehensions and `map` method of `Series` can also be used to produce more complex criteria:

```
In [153]: df2 = pd.DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six',
↳ ],
.....:                        'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
.....:                        'c' : np.random.randn(7)})
.....:
```

```
# only want 'two' or 'three'
```

```
In [154]: criterion = df2['a'].map(lambda x: x.startswith('t'))
```

```
In [155]: df2[criterion]
```

```
Out [155] :
```

|   | a     | b | c         |
|---|-------|---|-----------|
| 2 | two   | y | 0.041290  |
| 3 | three | x | 0.361719  |
| 4 | two   | y | -0.238075 |

```
# equivalent but slower
```

```
In [156]: df2[[x.startswith('t') for x in df2['a']]]
```

|   | a     | b | c         |
|---|-------|---|-----------|
| 2 | two   | y | 0.041290  |
| 3 | three | x | 0.361719  |
| 4 | two   | y | -0.238075 |

```
# Multiple criteria
```

```
In [157]: df2[criterion & (df2['b'] == 'x')]
```

|   | a     | b | c        |
|---|-------|---|----------|
| 3 | three | x | 0.361719 |

With the choice methods *Selection by Label*, *Selection by Position*, and *Advanced Indexing* you may select along more than one axis using boolean vectors combined with other indexing expressions.

```
In [158]: df2.loc[criterion & (df2['b'] == 'x'), 'b':'c']
```

```
Out[158]:
```

|     | b        | c |
|-----|----------|---|
| 3 x | 0.361719 |   |

## 12.14 Indexing with isin

Consider the `isin()` method of `Series`, which returns a boolean vector that is true wherever the `Series` elements exist in the passed list. This allows you to select rows where one or more columns have values you want:

```
In [159]: s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='int64')
```

```
In [160]: s
```

```
Out[160]:
4    0
3    1
2    2
1    3
0    4
dtype: int64
```

```
In [161]: s.isin([2, 4, 6])
```

```
Out[161]:
4    False
3    False
2     True
1    False
0     True
dtype: bool
```

```
In [162]: s[s.isin([2, 4, 6])]
```

```
Out[162]:
2    2
0    4
dtype: int64
```

The same method is available for Index objects and is useful for the cases when you don't know which of the sought labels are in fact present:

```
In [163]: s[s.index.isin([2, 4, 6])]
```

```
Out[163]:
4    0
2    2
dtype: int64
```

```
# compare it to the following
```

```
In [164]: s.reindex([2, 4, 6])
```

```
Out[164]:
2    2.0
4    0.0
6    NaN
dtype: float64
```

In addition to that, MultiIndex allows selecting a separate level to use in the membership check:

```
In [165]: s_mi = pd.Series(np.arange(6),
.....:                    index=pd.MultiIndex.from_product([[0, 1], ['a', 'b', 'c']],
.....:                    names=['level0', 'level1']))
```

```
In [166]: s_mi
```

```
Out[166]:
level0 level1  value
0     a      0      0
      b      1      1
      c      2      2
1     a      3      3
```

(continues on next page)

(continued from previous page)

```

    b    4
    c    5
dtype: int64

In [167]: s_mi.iloc[s_mi.index.isin([(1, 'a'), (2, 'b'), (0, 'c')])]
Out[167]:
↪
0  c    2
1  a    3
dtype: int64

In [168]: s_mi.iloc[s_mi.index.isin(['a', 'c', 'e'], level=1)]
Out[168]:
↪
0  a    0
   c    2
1  a    3
   c    5
dtype: int64

```

DataFrame also has an `isin()` method. When calling `isin`, pass a set of values as either an array or dict. If values is an array, `isin` returns a DataFrame of booleans that is the same shape as the original DataFrame, with True wherever the element is in the sequence of values.

```

In [169]: df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f', 'n'],
.....:                      'ids2': ['a', 'n', 'c', 'n']})
.....:

In [170]: values = ['a', 'b', 1, 3]

In [171]: df.isin(values)
Out[171]:
   vals  ids  ids2
0  True  True  True
1 False  True False
2  True False False
3 False False False

```

Oftentimes you'll want to match certain values with certain columns. Just make values a dict where the key is the column, and the value is a list of items you want to check for.

```

In [172]: values = {'ids': ['a', 'b'], 'vals': [1, 3]}

In [173]: df.isin(values)
Out[173]:
   vals  ids  ids2
0  True  True False
1 False  True False
2  True False False
3 False False False

```

Combine DataFrame's `isin` with the `any()` and `all()` methods to quickly select subsets of your data that meet a given criteria. To select a row where each column meets its own criterion:

```

In [174]: values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1, 3]}

```

(continues on next page)

(continued from previous page)

```
In [175]: row_mask = df.isin(values).all(1)
```

```
In [176]: df[row_mask]
```

```
Out[176]:
   vals ids ids2
0     1  a     a
```

## 12.15 The where () Method and Masking

Selecting values from a Series with a boolean vector generally returns a subset of the data. To guarantee that selection output has the same shape as the original data, you can use the `where` method in `Series` and `DataFrame`.

To return only the selected rows:

```
In [177]: s[s > 0]
```

```
Out[177]:
3     1
2     2
1     3
0     4
dtype: int64
```

To return a Series of the same shape as the original:

```
In [178]: s.where(s > 0)
```

```
Out[178]:
4     NaN
3     1.0
2     2.0
1     3.0
0     4.0
dtype: float64
```

Selecting values from a `DataFrame` with a boolean criterion now also preserves input data shape. `where` is used under the hood as the implementation. The code below is equivalent to `df[df < 0]`.

```
In [179]: df[df < 0]
```

```
Out[179]:
           A           B           C           D
2000-01-01 -2.104139 -1.309525        NaN        NaN
2000-01-02 -0.352480         NaN -1.192319        NaN
2000-01-03 -0.864883         NaN -0.227870        NaN
2000-01-04         NaN -1.222082         NaN -1.233203
2000-01-05         NaN -0.605656 -1.169184         NaN
2000-01-06         NaN -0.948458         NaN -0.684718
2000-01-07 -2.670153 -0.114722         NaN -0.048048
2000-01-08         NaN         NaN -0.048788 -0.808838
```

In addition, `where` takes an optional `other` argument for replacement of values where the condition is `False`, in the returned copy.

```
In [180]: df.where(df < 0, -df)
```

```
Out[180]:
           A           B           C           D
```

(continues on next page)

(continued from previous page)

```

2000-01-01 -2.104139 -1.309525 -0.485855 -0.245166
2000-01-02 -0.352480 -0.390389 -1.192319 -1.655824
2000-01-03 -0.864883 -0.299674 -0.227870 -0.281059
2000-01-04 -0.846958 -1.222082 -0.600705 -1.233203
2000-01-05 -0.669692 -0.605656 -1.169184 -0.342416
2000-01-06 -0.868584 -0.948458 -2.297780 -0.684718
2000-01-07 -2.670153 -0.114722 -0.168904 -0.048048
2000-01-08 -0.801196 -1.392071 -0.048788 -0.808838

```

You may wish to set values based on some boolean criteria. This can be done intuitively like so:

```

In [181]: s2 = s.copy()

In [182]: s2[s2 < 0] = 0

In [183]: s2
Out[183]:
4      0
3      1
2      2
1      3
0      4
dtype: int64

In [184]: df2 = df.copy()

In [185]: df2[df2 < 0] = 0

In [186]: df2
Out[186]:
              A              B              C              D
2000-01-01  0.000000  0.000000  0.485855  0.245166
2000-01-02  0.000000  0.390389  0.000000  1.655824
2000-01-03  0.000000  0.299674  0.000000  0.281059
2000-01-04  0.846958  0.000000  0.600705  0.000000
2000-01-05  0.669692  0.000000  0.000000  0.342416
2000-01-06  0.868584  0.000000  2.297780  0.000000
2000-01-07  0.000000  0.000000  0.168904  0.000000
2000-01-08  0.801196  1.392071  0.000000  0.000000

```

By default, `where` returns a modified copy of the data. There is an optional parameter `inplace` so that the original data can be modified without creating a copy:

```

In [187]: df_orig = df.copy()

In [188]: df_orig.where(df > 0, -df, inplace=True);

In [189]: df_orig
Out[189]:
              A              B              C              D
2000-01-01  2.104139  1.309525  0.485855  0.245166
2000-01-02  0.352480  0.390389  1.192319  1.655824
2000-01-03  0.864883  0.299674  0.227870  0.281059
2000-01-04  0.846958  1.222082  0.600705  1.233203
2000-01-05  0.669692  0.605656  1.169184  0.342416
2000-01-06  0.868584  0.948458  2.297780  0.684718

```

(continues on next page)

(continued from previous page)

|            |          |          |          |          |
|------------|----------|----------|----------|----------|
| 2000-01-07 | 2.670153 | 0.114722 | 0.168904 | 0.048048 |
| 2000-01-08 | 0.801196 | 1.392071 | 0.048788 | 0.808838 |

**Note:** The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

```
In [190]: df.where(df < 0, -df) == np.where(df < 0, df, -df)
```

```
Out [190]:
```

|            | A    | B    | C    | D    |
|------------|------|------|------|------|
| 2000-01-01 | True | True | True | True |
| 2000-01-02 | True | True | True | True |
| 2000-01-03 | True | True | True | True |
| 2000-01-04 | True | True | True | True |
| 2000-01-05 | True | True | True | True |
| 2000-01-06 | True | True | True | True |
| 2000-01-07 | True | True | True | True |
| 2000-01-08 | True | True | True | True |

### alignment

Furthermore, `where` aligns the input boolean condition (ndarray or DataFrame), such that partial selection with setting is possible. This is analogous to partial setting via `.loc` (but on the contents rather than the axis labels).

```
In [191]: df2 = df.copy()
```

```
In [192]: df2[df2[1:4] > 0] = 3
```

```
In [193]: df2
```

```
Out [193]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | -2.104139 | -1.309525 | 0.485855  | 0.245166  |
| 2000-01-02 | -0.352480 | 3.000000  | -1.192319 | 3.000000  |
| 2000-01-03 | -0.864883 | 3.000000  | -0.227870 | 3.000000  |
| 2000-01-04 | 3.000000  | -1.222082 | 3.000000  | -1.233203 |
| 2000-01-05 | 0.669692  | -0.605656 | -1.169184 | 0.342416  |
| 2000-01-06 | 0.868584  | -0.948458 | 2.297780  | -0.684718 |
| 2000-01-07 | -2.670153 | -0.114722 | 0.168904  | -0.048048 |
| 2000-01-08 | 0.801196  | 1.392071  | -0.048788 | -0.808838 |

Where can also accept `axis` and `level` parameters to align the input when performing the `where`.

```
In [194]: df2 = df.copy()
```

```
In [195]: df2.where(df2>0,df2['A'],axis='index')
```

```
Out [195]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | -2.104139 | -2.104139 | 0.485855  | 0.245166  |
| 2000-01-02 | -0.352480 | 0.390389  | -0.352480 | 1.655824  |
| 2000-01-03 | -0.864883 | 0.299674  | -0.864883 | 0.281059  |
| 2000-01-04 | 0.846958  | 0.846958  | 0.600705  | 0.846958  |
| 2000-01-05 | 0.669692  | 0.669692  | 0.669692  | 0.342416  |
| 2000-01-06 | 0.868584  | 0.868584  | 2.297780  | 0.868584  |
| 2000-01-07 | -2.670153 | -2.670153 | 0.168904  | -2.670153 |
| 2000-01-08 | 0.801196  | 1.392071  | 0.801196  | 0.801196  |

This is equivalent to (but faster than) the following.

```
In [196]: df2 = df.copy()

In [197]: df.apply(lambda x, y: x.where(x>0,y), y=df['A'])
Out[197]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | -2.104139 | -2.104139 | 0.485855  | 0.245166  |
| 2000-01-02 | -0.352480 | 0.390389  | -0.352480 | 1.655824  |
| 2000-01-03 | -0.864883 | 0.299674  | -0.864883 | 0.281059  |
| 2000-01-04 | 0.846958  | 0.846958  | 0.600705  | 0.846958  |
| 2000-01-05 | 0.669692  | 0.669692  | 0.669692  | 0.342416  |
| 2000-01-06 | 0.868584  | 0.868584  | 2.297780  | 0.868584  |
| 2000-01-07 | -2.670153 | -2.670153 | 0.168904  | -2.670153 |
| 2000-01-08 | 0.801196  | 1.392071  | 0.801196  | 0.801196  |

New in version 0.18.1.

Where can accept a callable as condition and other arguments. The function must be with one argument (the calling Series or DataFrame) and that returns valid output as condition and other argument.

```
In [198]: df3 = pd.DataFrame({'A': [1, 2, 3],
.....:                      'B': [4, 5, 6],
.....:                      'C': [7, 8, 9]})
.....:

In [199]: df3.where(lambda x: x > 4, lambda x: x + 10)
Out[199]:
```

|   | A  | B  | C |
|---|----|----|---|
| 0 | 11 | 14 | 7 |
| 1 | 12 | 5  | 8 |
| 2 | 13 | 6  | 9 |

## 12.15.1 Mask

`mask()` is the inverse boolean operation of `where`.

```
In [200]: s.mask(s >= 0)
Out[200]:
```

|   |     |
|---|-----|
| 4 | NaN |
| 3 | NaN |
| 2 | NaN |
| 1 | NaN |
| 0 | NaN |

dtype: float64

```
In [201]: df.mask(df >= 0)
Out[201]:
```

|            | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| 2000-01-01 | -2.104139 | -1.309525 | NaN       | NaN       |
| 2000-01-02 | -0.352480 | NaN       | -1.192319 | NaN       |
| 2000-01-03 | -0.864883 | NaN       | -0.227870 | NaN       |
| 2000-01-04 | NaN       | -1.222082 | NaN       | -1.233203 |
| 2000-01-05 | NaN       | -0.605656 | -1.169184 | NaN       |
| 2000-01-06 | NaN       | -0.948458 | NaN       | -0.684718 |
| 2000-01-07 | -2.670153 | -0.114722 | NaN       | -0.048048 |
| 2000-01-08 | NaN       | NaN       | -0.048788 | -0.808838 |



## 12.16 The `query()` Method

`DataFrame` objects have a `query()` method that allows selection using an expression.

You can get the value of the frame where column `b` has values between the values of columns `a` and `c`. For example:

```
In [202]: n = 10

In [203]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))

In [204]: df
Out[204]:
```

|   | a        | b        | c        |
|---|----------|----------|----------|
| 0 | 0.438921 | 0.118680 | 0.863670 |
| 1 | 0.138138 | 0.577363 | 0.686602 |
| 2 | 0.595307 | 0.564592 | 0.520630 |
| 3 | 0.913052 | 0.926075 | 0.616184 |
| 4 | 0.078718 | 0.854477 | 0.898725 |
| 5 | 0.076404 | 0.523211 | 0.591538 |
| 6 | 0.792342 | 0.216974 | 0.564056 |
| 7 | 0.397890 | 0.454131 | 0.915716 |
| 8 | 0.074315 | 0.437913 | 0.019794 |
| 9 | 0.559209 | 0.502065 | 0.026437 |

```
# pure python
In [205]: df[(df.a < df.b) & (df.b < df.c)]
///////////////////////////////////////////////////
↪
```

|   | a        | b        | c        |
|---|----------|----------|----------|
| 1 | 0.138138 | 0.577363 | 0.686602 |
| 4 | 0.078718 | 0.854477 | 0.898725 |
| 5 | 0.076404 | 0.523211 | 0.591538 |
| 7 | 0.397890 | 0.454131 | 0.915716 |

```
# query
In [206]: df.query('(a < b) & (b < c)')
///////////////////////////////////////////////////
↪
```

|   | a        | b        | c        |
|---|----------|----------|----------|
| 1 | 0.138138 | 0.577363 | 0.686602 |
| 4 | 0.078718 | 0.854477 | 0.898725 |
| 5 | 0.076404 | 0.523211 | 0.591538 |
| 7 | 0.397890 | 0.454131 | 0.915716 |

Do the same thing but fall back on a named index if there is no column with the name `a`.

```
In [207]: df = pd.DataFrame(np.random.randint(n / 2, size=(n, 2)), columns=list('bc'))

In [208]: df.index.name = 'a'

In [209]: df
Out[209]:
```

| a | b | c |
|---|---|---|
| 0 | 0 | 4 |
| 1 | 0 | 1 |
| 2 | 3 | 4 |

(continues on next page)

(continued from previous page)

```

3  4  3
4  1  4
5  0  3
6  0  1
7  3  4
8  2  3
9  1  1

```

```
In [210]: df.query('a < b and b < c')
```

```

////////////////////////////////////
↪
   b  c
a
2  3  4

```

If instead you don't want to or cannot name your index, you can use the name `index` in your query expression:

```
In [211]: df = pd.DataFrame(np.random.randint(n, size=(n, 2)), columns=list('bc'))
```

```
In [212]: df
```

```
Out[212]:
```

```

   b  c
0  3  1
1  3  0
2  5  6
3  5  2
4  7  4
5  0  1
6  2  5
7  0  1
8  6  0
9  7  9

```

```
In [213]: df.query('index < b < c')
```

```

////////////////////////////////////
↪
   b  c
2  5  6

```

**Note:** If the name of your index overlaps with a column name, the column name is given precedence. For example,

```
In [214]: df = pd.DataFrame({'a': np.random.randint(5, size=5)})
```

```
In [215]: df.index.name = 'a'
```

```
In [216]: df.query('a > 2') # uses the column 'a', not the index
```

```
Out[216]:
```

```

   a
a
1  3
3  3

```

You can still use the index in a query expression by using the special identifier `'index'`:

```
In [217]: df.query('index > 2')
```

(continues on next page)

(continued from previous page)

**Out [217]:**

```

a
a
3  3
4  2

```

If for some reason you have a column named `index`, then you can refer to the index as `ilevel_0` as well, but at this point you should consider renaming your columns to something less ambiguous.

### 12.16.1 MultiIndex query() Syntax

You can also use the levels of a DataFrame with a *MultiIndex* as if they were columns in the frame:

**In [218]:** `n = 10`**In [219]:** `colors = np.random.choice(['red', 'green'], size=n)`**In [220]:** `foods = np.random.choice(['eggs', 'ham'], size=n)`**In [221]:** `colors`**Out [221]:**

```

array(['red', 'red', 'red', 'green', 'green', 'green', 'green', 'green',
       'green', 'green'],
      dtype='<U5')

```

**In [222]:** `foods`

```

////////////////////////////////////
↪
array(['ham', 'ham', 'eggs', 'eggs', 'eggs', 'ham', 'ham', 'eggs', 'eggs',
       'eggs'],
      dtype='<U4')

```

**In [223]:** `index = pd.MultiIndex.from_arrays([colors, foods], names=['color', 'food'])`**In [224]:** `df = pd.DataFrame(np.random.randn(n, 2), index=index)`**In [225]:** `df`**Out [225]:**

```

           0           1
color food
red  ham    0.194889 -0.381994
     ham    0.318587  2.089075
     eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
     eggs -2.029766  0.792652
     ham    0.461007 -0.542749
     ham   -0.305384 -0.479195
     eggs  0.095031 -0.270099
     eggs -0.707140 -0.773882
     eggs  0.229453  0.304418

```

**In [226]:** `df.query('color == "red"')`

```

////////////////////////////////////
↪

```

(continues on next page)

(continued from previous page)

```

           0          1
color food
red  ham    0.194889 -0.381994
     ham    0.318587  2.089075
     eggs -0.728293 -0.090255

```

If the levels of the `MultiIndex` are unnamed, you can refer to them using special names:

```
In [227]: df.index.names = [None, None]
```

```
In [228]: df
```

```
Out [228]:
           0          1
red  ham    0.194889 -0.381994
     ham    0.318587  2.089075
     eggs -0.728293 -0.090255
green eggs -0.748199  1.318931
     eggs -2.029766  0.792652
     ham    0.461007 -0.542749
     ham   -0.305384 -0.479195
     eggs  0.095031 -0.270099
     eggs -0.707140 -0.773882
     eggs  0.229453  0.304418

```

```
In [229]: df.query('ilevel_0 == "red"')
```

```

////////////////////////////////////
↪
           0          1
red ham    0.194889 -0.381994
     ham    0.318587  2.089075
     eggs -0.728293 -0.090255

```

The convention is `ilevel_0`, which means “index level 0” for the 0th level of the index.

## 12.16.2 `query()` Use Cases

A use case for `query()` is when you have a collection of `DataFrame` objects that have a subset of column names (or index levels/names) in common. You can pass the same query to both frames *without* having to specify which frame you’re interested in querying

```
In [230]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))
```

```
In [231]: df
```

```
Out [231]:
           a          b          c
0  0.224283  0.736107  0.139168
1  0.302827  0.657803  0.713897
2  0.611185  0.136624  0.984960
3  0.195246  0.123436  0.627712
4  0.618673  0.371660  0.047902
5  0.480088  0.062993  0.185760
6  0.568018  0.483467  0.445289
7  0.309040  0.274580  0.587101
8  0.258993  0.477769  0.370255
9  0.550459  0.840870  0.304611

```

(continues on next page)

(continued from previous page)

```
In [232]: df2 = pd.DataFrame(np.random.rand(n + 2, 3), columns=df.columns)
```

```
In [233]: df2
```

```
Out[233]:
```

|    | a        | b        | c        |
|----|----------|----------|----------|
| 0  | 0.357579 | 0.229800 | 0.596001 |
| 1  | 0.309059 | 0.957923 | 0.965663 |
| 2  | 0.123102 | 0.336914 | 0.318616 |
| 3  | 0.526506 | 0.323321 | 0.860813 |
| 4  | 0.518736 | 0.486514 | 0.384724 |
| 5  | 0.190804 | 0.505723 | 0.614533 |
| 6  | 0.891939 | 0.623977 | 0.676639 |
| 7  | 0.480559 | 0.378528 | 0.460858 |
| 8  | 0.420223 | 0.136404 | 0.141295 |
| 9  | 0.732206 | 0.419540 | 0.604675 |
| 10 | 0.604466 | 0.848974 | 0.896165 |
| 11 | 0.589168 | 0.920046 | 0.732716 |

```
In [234]: expr = '0.0 <= a <= c <= 0.5'
```

```
In [235]: map(lambda frame: frame.query(expr), [df, df2])
```

```
Out[235]: <map at 0x1c2b11f5f8>
```

### 12.16.3 query() Python versus pandas Syntax Comparison

Full numpy-like syntax:

```
In [236]: df = pd.DataFrame(np.random.randint(n, size=(n, 3)), columns=list('abc'))
```

```
In [237]: df
```

```
Out[237]:
```

|   | a | b | c |
|---|---|---|---|
| 0 | 7 | 8 | 9 |
| 1 | 1 | 0 | 7 |
| 2 | 2 | 7 | 2 |
| 3 | 6 | 2 | 2 |
| 4 | 2 | 6 | 3 |
| 5 | 3 | 8 | 2 |
| 6 | 1 | 7 | 2 |
| 7 | 5 | 1 | 5 |
| 8 | 9 | 8 | 0 |
| 9 | 1 | 5 | 0 |

```
In [238]: df.query('(a < b) & (b < c)')
```

```

////////////////////////////////////
↪
   a  b  c
0  7  8  9

```

```
In [239]: df[(df.a < df.b) & (df.b < df.c)]
```

```

////////////////////////////////////
↪
   a  b  c
0  7  8  9

```

Slightly nicer by removing the parentheses (by binding making comparison operators bind tighter than `&` and `|`).

```
In [240]: df.query('a < b & b < c')
Out[240]:
```

|   | a | b | c |
|---|---|---|---|
| 0 | 7 | 8 | 9 |

Use English instead of symbols:

```
In [241]: df.query('a < b and b < c')
Out[241]:
```

|   | a | b | c |
|---|---|---|---|
| 0 | 7 | 8 | 9 |

Pretty close to how you might write it on paper:

```
In [242]: df.query('a < b < c')
Out[242]:
```

|   | a | b | c |
|---|---|---|---|
| 0 | 7 | 8 | 9 |

#### 12.16.4 The `in` and `not in` operators

`query()` also supports special use of Python's `in` and `not in` comparison operators, providing a succinct syntax for calling the `isin` method of a `Series` or `DataFrame`.

```
# get all rows where columns "a" and "b" have overlapping values
In [243]: df = pd.DataFrame({'a': list('aabbccddeeff'), 'b': list('aaaabbbbbcccc'),
.....:                       'c': np.random.randint(5, size=12),
.....:                       'd': np.random.randint(9, size=12)})
```

```
In [244]: df
Out[244]:
```

|    | a | b | c | d |
|----|---|---|---|---|
| 0  | a | a | 2 | 6 |
| 1  | a | a | 4 | 7 |
| 2  | b | a | 1 | 6 |
| 3  | b | a | 2 | 1 |
| 4  | c | b | 3 | 6 |
| 5  | c | b | 0 | 2 |
| 6  | d | b | 3 | 3 |
| 7  | d | b | 2 | 1 |
| 8  | e | c | 4 | 3 |
| 9  | e | c | 2 | 0 |
| 10 | f | c | 0 | 6 |
| 11 | f | c | 1 | 2 |

```
In [245]: df.query('a in b')
```

|   | a | b | c | d |
|---|---|---|---|---|
| 0 | a | a | 2 | 6 |
| 1 | a | a | 4 | 7 |
| 2 | b | a | 1 | 6 |
| 3 | b | a | 2 | 1 |

(continues on next page)

(continued from previous page)

```
4 c b 3 6
5 c b 0 2
```

```
# How you'd do it in pure Python
```

```
In [246]: df[df.a.isin(df.b)]
```

```

a b c d
0 a a 2 6
1 a a 4 7
2 b a 1 6
3 b a 2 1
4 c b 3 6
5 c b 0 2
```

```
In [247]: df.query('a not in b')
```

```

a b c d
6 d b 3 3
7 d b 2 1
8 e c 4 3
9 e c 2 0
10 f c 0 6
11 f c 1 2
```

```
# pure Python
```

```
In [248]: df[~df.a.isin(df.b)]
```

```

a b c d
6 d b 3 3
7 d b 2 1
8 e c 4 3
9 e c 2 0
10 f c 0 6
11 f c 1 2
```

You can combine this with other expressions for very succinct queries:

```
# rows where cols a and b have overlapping values and col c's values are less than_
↳ col d's
```

```
In [249]: df.query('a in b and c < d')
```

```
Out[249]:
```

```

a b c d
0 a a 2 6
1 a a 4 7
2 b a 1 6
4 c b 3 6
5 c b 0 2
```

```
# pure Python
```

```
In [250]: df[df.b.isin(df.a) & (df.c < df.d)]
```

```

a b c d
0 a a 2 6
```

Out [250]:

(continues on next page)

(continued from previous page)

|    |   |   |   |   |
|----|---|---|---|---|
| 1  | a | a | 4 | 7 |
| 2  | b | a | 1 | 6 |
| 4  | c | b | 3 | 6 |
| 5  | c | b | 0 | 2 |
| 10 | f | c | 0 | 6 |
| 11 | f | c | 1 | 2 |

**Note:** Note that `in` and `not in` are evaluated in Python, since `numexpr` has no equivalent of this operation. However, **only the `in/not in` expression itself** is evaluated in vanilla Python. For example, in the expression

```
df.query('a in b + c + d')
```

`(b + c + d)` is evaluated by `numexpr` and *then* the `in` operation is evaluated in plain Python. In general, any operations that can be evaluated using `numexpr` will be.

## 12.16.5 Special use of the `==` operator with `list` objects

Comparing a `list` of values to a column using `==/!=` works similarly to `in/not in`.

```
In [251]: df.query('b == ["a", "b", "c"]')
```

```
Out[251]:
```

|    | a | b | c | d |
|----|---|---|---|---|
| 0  | a | a | 2 | 6 |
| 1  | a | a | 4 | 7 |
| 2  | b | a | 1 | 6 |
| 3  | b | a | 2 | 1 |
| 4  | c | b | 3 | 6 |
| 5  | c | b | 0 | 2 |
| 6  | d | b | 3 | 3 |
| 7  | d | b | 2 | 1 |
| 8  | e | c | 4 | 3 |
| 9  | e | c | 2 | 0 |
| 10 | f | c | 0 | 6 |
| 11 | f | c | 1 | 2 |

```
# pure Python
```

```
In [252]: df[df.b.isin(["a", "b", "c"])]
```

```

////////////////////////////////////
↪

```

|    | a | b | c | d |
|----|---|---|---|---|
| 0  | a | a | 2 | 6 |
| 1  | a | a | 4 | 7 |
| 2  | b | a | 1 | 6 |
| 3  | b | a | 2 | 1 |
| 4  | c | b | 3 | 6 |
| 5  | c | b | 0 | 2 |
| 6  | d | b | 3 | 3 |
| 7  | d | b | 2 | 1 |
| 8  | e | c | 4 | 3 |
| 9  | e | c | 2 | 0 |
| 10 | f | c | 0 | 6 |
| 11 | f | c | 1 | 2 |

(continues on next page)



(continued from previous page)

```
In [253]: df.query('c == [1, 2]')
```

```

=====
↪
   a  b  c  d
0  a  a  2  6
2  b  a  1  6
3  b  a  2  1
7  d  b  2  1
9  e  c  2  0
11 f  c  1  2

```

```
In [254]: df.query('c != [1, 2]')
```

```

=====
↪
   a  b  c  d
1  a  a  4  7
4  c  b  3  6
5  c  b  0  2
6  d  b  3  3
8  e  c  4  3
10 f  c  0  6

```

```
# using in/not in
```

```
In [255]: df.query('[1, 2] in c')
```

```

=====
↪
   a  b  c  d
0  a  a  2  6
2  b  a  1  6
3  b  a  2  1
7  d  b  2  1
9  e  c  2  0
11 f  c  1  2

```

```
In [256]: df.query('[1, 2] not in c')
```

```

=====
↪
   a  b  c  d
1  a  a  4  7
4  c  b  3  6
5  c  b  0  2
6  d  b  3  3
8  e  c  4  3
10 f  c  0  6

```

```
# pure Python
```

```
In [257]: df[df.c.isin([1, 2])]
```

```

=====
↪
   a  b  c  d
0  a  a  2  6
2  b  a  1  6
3  b  a  2  1
7  d  b  2  1
9  e  c  2  0
11 f  c  1  2

```

## 12.16.6 Boolean Operators

You can negate boolean expressions with the word `not` or the `~` operator.

```
In [258]: df = pd.DataFrame(np.random.rand(n, 3), columns=list('abc'))
```

```
In [259]: df['bools'] = np.random.rand(len(df)) > 0.5
```

```
In [260]: df.query('~bools')
```

```
Out[260]:
```

|   | a        | b        | c        | bools |
|---|----------|----------|----------|-------|
| 2 | 0.697753 | 0.212799 | 0.329209 | False |
| 7 | 0.275396 | 0.691034 | 0.826619 | False |
| 8 | 0.190649 | 0.558748 | 0.262467 | False |

```
In [261]: df.query('not bools')
```

```
Out[261]:
```

|   | a        | b        | c        | bools |
|---|----------|----------|----------|-------|
| 2 | 0.697753 | 0.212799 | 0.329209 | False |
| 7 | 0.275396 | 0.691034 | 0.826619 | False |
| 8 | 0.190649 | 0.558748 | 0.262467 | False |

```
In [262]: df.query('not bools') == df[~df.bools]
```

```
Out[262]:
```

|   | a    | b    | c    | bools |
|---|------|------|------|-------|
| 2 | True | True | True | True  |
| 7 | True | True | True | True  |
| 8 | True | True | True | True  |

Of course, expressions can be arbitrarily complex too:

```
# short query syntax
```

```
In [263]: shorter = df.query('a < b < c and (not bools) or bools > 2')
```

```
# equivalent in pure Python
```

```
In [264]: longer = df[(df.a < df.b) & (df.b < df.c) & (~df.bools) | (df.bools > 2)]
```

```
In [265]: shorter
```

```
Out[265]:
```

|   | a        | b        | c        | bools |
|---|----------|----------|----------|-------|
| 7 | 0.275396 | 0.691034 | 0.826619 | False |

```
In [266]: longer
```

```
Out[266]:
```

|   | a        | b        | c        | bools |
|---|----------|----------|----------|-------|
| 7 | 0.275396 | 0.691034 | 0.826619 | False |

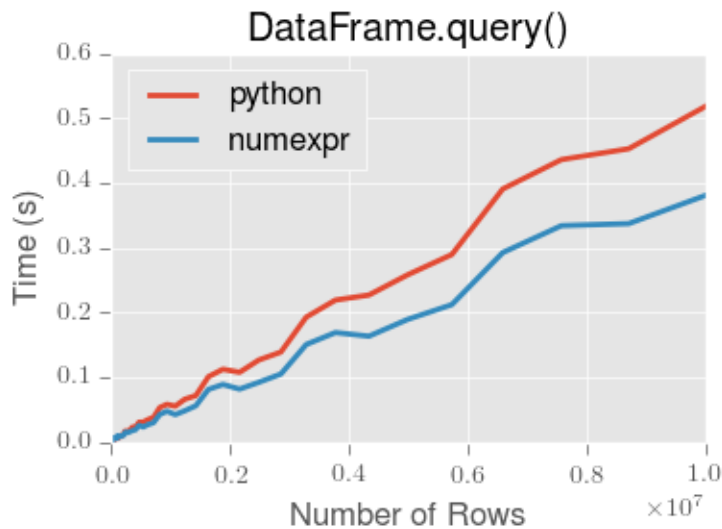
```
In [267]: shorter == longer
```

```
Out[267]:
```

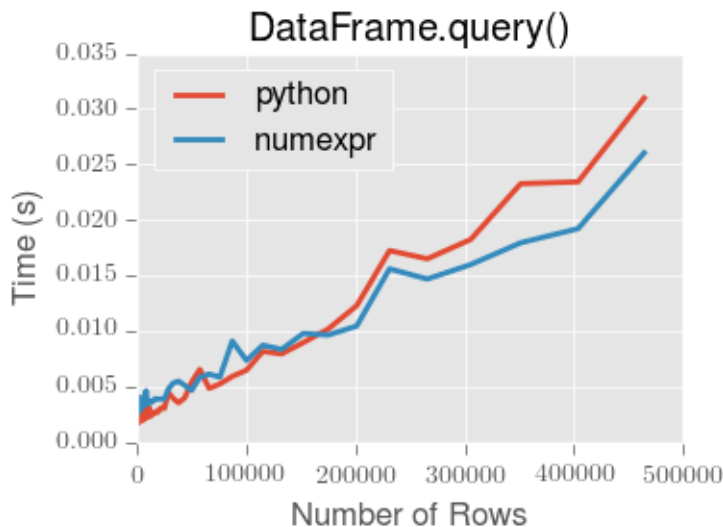
|   | a    | b    | c    | bools |
|---|------|------|------|-------|
| 7 | True | True | True | True  |

### 12.16.7 Performance of `query()`

`DataFrame.query()` using `numexpr` is slightly faster than Python for large frames.



**Note:** You will only see the performance benefits of using the `numexpr` engine with `DataFrame.query()` if your frame has more than approximately 200,000 rows.



This plot was created using a `DataFrame` with 3 columns each containing floating point values generated using `numpy.random.randn()`.

## 12.17 Duplicate Data

If you want to identify and remove duplicate rows in a `DataFrame`, there are two methods that will help: `duplicated` and `drop_duplicates`. Each takes as an argument the columns to use to identify duplicated rows.

- `df.duplicated()` returns a boolean vector whose length is the number of rows, and which indicates whether a row is duplicated.
- `df.drop_duplicates()` removes duplicate rows.

By default, the first observed row of a duplicate set is considered unique, but each method has a `keep` parameter to specify targets to be kept.

- `keep='first'` (default): mark / drop duplicates except for the first occurrence.
- `keep='last'`: mark / drop duplicates except for the last occurrence.
- `keep=False`: mark / drop all duplicates.

```
In [268]: df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two', 'three', 'four']
→      ,
      .....:                      'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
      .....:                      'c': np.random.randn(7)})
      .....:
```

```
In [269]: df2
```

```
Out[269]:
```

|   | a     | b | c         |
|---|-------|---|-----------|
| 0 | one   | x | -1.067137 |
| 1 | one   | y | 0.309500  |
| 2 | two   | x | -0.211056 |
| 3 | two   | y | -1.842023 |
| 4 | two   | x | -0.390820 |
| 5 | three | x | -1.964475 |
| 6 | four  | x | 1.298329  |

```
In [270]: df2.duplicated('a')
```

```

→
0    False
1     True
2    False
3     True
4     True
5    False
6    False
dtype: bool
```

```
In [271]: df2.duplicated('a', keep='last')
```

```

→
0     True
1    False
2     True
3     True
4    False
5    False
6    False
dtype: bool
```

```
In [272]: df2.duplicated('a', keep=False)
```

```

→
0     True
1     True
```

(continues on next page)

(continued from previous page)

```

2     True
3     True
4     True
5    False
6    False
dtype: bool

```

```
In [273]: df2.drop_duplicates('a')
```

```

↪
      a  b      c
0  one  x -1.067137
2  two  x -0.211056
5 three  x -1.964475
6  four  x  1.298329

```

```
In [274]: df2.drop_duplicates('a', keep='last')
```

```

↪
      a  b      c
1  one  y  0.309500
4  two  x -0.390820
5 three  x -1.964475
6  four  x  1.298329

```

```
In [275]: df2.drop_duplicates('a', keep=False)
```

```

↪
      a  b      c
5 three  x -1.964475
6  four  x  1.298329

```

Also, you can pass a list of columns to identify duplications.

```
In [276]: df2.duplicated(['a', 'b'])
```

```

Out[276]:
0    False
1    False
2    False
3    False
4     True
5    False
6    False
dtype: bool

```

```
In [277]: df2.drop_duplicates(['a', 'b'])
```

```

↪
      a  b      c
0  one  x -1.067137
1  one  y  0.309500
2  two  x -0.211056
3  two  y -1.842023
5 three  x -1.964475
6  four  x  1.298329

```

To drop duplicates by index value, use `Index.duplicated` then perform slicing. The same set of options are

available for the `keep` parameter.

```
In [278]: df3 = pd.DataFrame({'a': np.arange(6),  
.....:                      'b': np.random.randn(6)},  
.....:                      index=['a', 'a', 'b', 'c', 'b', 'a'])  
.....:
```

```
In [279]: df3
```

Out [279] :

|   | a | b         |
|---|---|-----------|
| a | 0 | 1.440455  |
| a | 1 | 2.456086  |
| b | 2 | 1.038402  |
| c | 3 | -0.894409 |
| b | 4 | 0.683536  |
| a | 5 | 3.082764  |

```
In [280]: df3.index.duplicated()
```

```

\\array([False,  True, False, False,  True,  True], dtype=bool)

```

```
In [281]: df3[~df3.index.duplicated()]
```

$\backslash \backslash \backslash \backslash \backslash \backslash \backslash \backslash \backslash \backslash \backslash \backslash$

$\leftrightarrow$

|   | a | b         |
|---|---|-----------|
| a | 0 | 1.440455  |
| b | 2 | 1.038402  |
| c | 3 | -0.894409 |

```
In [282]: df3[~df3.index.duplicated(keep='last')]
```

|   | a | b         |
|---|---|-----------|
| c | 3 | -0.894409 |
| b | 4 | 0.683536  |
| a | 5 | 3.082764  |

```
In [283]: df3[~df3.index.duplicated(keep=False)]
```


  
 a                      b
   
 c    3   -0.894409

## 12.18 Dictionary-like `get()` method

Each of Series, DataFrame, and Panel have a `get` method which can return a default value.

```
In [284]: s = pd.Series([1,2,3], index=['a','b','c'])
```

```
In [285]: s.get('a') # equivalent to s['a']
```

Out [285]: 1

```
In [286]: s.get('x', default=-1)
```

```
Out[286]: -1
```

## 12.19 The `lookup()` Method

Sometimes you want to extract a set of values given a sequence of row labels and column labels, and the `lookup` method allows for this and returns a NumPy array. For instance:

```
In [287]: dflookup = pd.DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])

In [288]: dflookup.lookup(list(range(0,10,2)), ['B','C','A','B','D'])
Out[288]: array([ 0.3506,  0.4779,  0.4825,  0.9197,  0.5019])
```

## 12.20 Index objects

The pandas *Index* class and its subclasses can be viewed as implementing an *ordered multiset*. Duplicates are allowed. However, if you try to convert an *Index* object with duplicate entries into a set, an exception will be raised.

*Index* also provides the infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create an *Index* directly is to pass a list or other sequence to *Index*:

```
In [289]: index = pd.Index(['e', 'd', 'a', 'b'])

In [290]: index
Out[290]: Index(['e', 'd', 'a', 'b'], dtype='object')

In [291]: 'd' in index
Out[291]: True
```

You can also pass a name to be stored in the index:

```
In [292]: index = pd.Index(['e', 'd', 'a', 'b'], name='something')

In [293]: index.name
Out[293]: 'something'
```

The name, if set, will be shown in the console display:

```
In [294]: index = pd.Index(list(range(5)), name='rows')

In [295]: columns = pd.Index(['A', 'B', 'C'], name='cols')

In [296]: df = pd.DataFrame(np.random.randn(5, 3), index=index, columns=columns)

In [297]: df
Out[297]:
cols      A      B      C
rows
0      1.295989  0.185778  0.436259
1      0.678101  0.311369 -0.528378
2     -0.674808 -1.103529 -0.656157
3      1.889957  2.076651 -1.102192
4     -1.211795 -0.791746  0.634724

In [298]: df['A']
```

(continues on next page)

(continued from previous page)

```

rows
0      1.295989
1      0.678101
2     -0.674808
3      1.889957
4     -1.211795
Name: A, dtype: float64

```

### 12.20.1 Setting metadata

Indexes are “mostly immutable”, but it is possible to set and change their metadata, like the index name (or, for MultiIndex, levels and labels).

You can use the `rename`, `set_names`, `set_levels`, and `set_labels` to set these attributes directly. They default to returning a copy; however, you can specify `inplace=True` to have the data change in place.

See *Advanced Indexing* for usage of MultiIndexes.

```

In [299]: ind = pd.Index([1, 2, 3])

In [300]: ind.rename("apple")
Out[300]: Int64Index([1, 2, 3], dtype='int64', name='apple')

In [301]: ind
Out[301]: Int64Index([1, 2, 3], dtype='int64')

In [302]: ind.set_names(["apple"], inplace=True)

In [303]: ind.name = "bob"

In [304]: ind
Out[304]: Int64Index([1, 2, 3], dtype='int64', name='bob')

```

`set_names`, `set_levels`, and `set_labels` also take an optional *level* argument

```

In [305]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first',
Out[305]: MultiIndex
Out[306]: MultiIndex(levels=[[0, 1, 2], ['one', 'two']],
labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
names=['first', 'second'])

In [307]: index.levels[1]
Out[307]: Index(['one', 'two'], dtype='object', name='second')

In [308]: index.set_levels(["a", "b"], level=1)
Out[308]: MultiIndex(levels=[[0, 1, 2], ['a', 'b']],
labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
names=['first', 'second'])

```



## 12.20.2 Set operations on Index objects

The two main operations are `union (|)` and `intersection (&)`. These can be directly called as instance methods or used via overloaded operators. Difference is provided via the `.difference()` method.

```
In [309]: a = pd.Index(['c', 'b', 'a'])

In [310]: b = pd.Index(['c', 'e', 'd'])

In [311]: a | b
Out[311]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')

In [312]: a & b
Out[312]: Index(['c'], dtype='object')

In [313]: a.difference(b)
Out[313]: Index(['a', 'b'], dtype='object')
```

Also available is the `symmetric_difference (^)` operation, which returns elements that appear in either `idx1` or `idx2`, but not in both. This is equivalent to the Index created by `idx1.difference(idx2).union(idx2.difference(idx1))`, with duplicates dropped.

```
In [314]: idx1 = pd.Index([1, 2, 3, 4])

In [315]: idx2 = pd.Index([2, 3, 4, 5])

In [316]: idx1.symmetric_difference(idx2)
Out[316]: Int64Index([1, 5], dtype='int64')

In [317]: idx1 ^ idx2
Out[317]: Int64Index([1, 5], dtype='int64')
```

**Note:** The resulting index from a set operation will be sorted in ascending order.

## 12.20.3 Missing values

**Important:** Even though `Index` can hold missing values (`NaN`), it should be avoided if you do not want any unexpected results. For example, some operations exclude missing values implicitly.

`Index.fillna` fills missing values with specified scalar value.

```
In [318]: idx1 = pd.Index([1, np.nan, 3, 4])

In [319]: idx1
Out[319]: Float64Index([1.0, nan, 3.0, 4.0], dtype='float64')

In [320]: idx1.fillna(2)
Out[320]: Float64Index([1.0, 2.0, 3.0, 4.0], dtype='float64')
```

(continues on next page)

(continued from previous page)

```

In [321]: idx2 = pd.DatetimeIndex([pd.Timestamp('2011-01-01'), pd.NaT, pd.Timestamp(
↳ '2011-01-03')])

In [322]: idx2
Out[322]: DatetimeIndex(['2011-01-01', 'NaT', '2011-01-03'], dtype='datetime64[ns]',
↳ freq=None)

In [323]: idx2.fillna(pd.Timestamp('2011-01-02'))
Out[323]: DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03'], dtype='datetime64[ns]',
↳ freq=None)

```

## 12.21 Set / Reset Index

Occasionally you will load or create a data set into a DataFrame and want to add an index after you've already done so. There are a couple of different ways.

### 12.21.1 Set an index

DataFrame has a `set_index()` method which takes a column name (for a regular Index) or a list of column names (for a MultiIndex). To create a new, re-indexed DataFrame:

```

In [324]: data
Out[324]:
   a  b  c  d
0 bar one z  1.0
1 bar two y  2.0
2 foo one x  3.0
3 foo two w  4.0

In [325]: indexed1 = data.set_index('c')

In [326]: indexed1
Out[326]:
   a  b  d
c
z bar one  1.0
y bar two  2.0
x foo one  3.0
w foo two  4.0

In [327]: indexed2 = data.set_index(['a', 'b'])

In [328]: indexed2
Out[328]:
   c  d
a  b
bar one z  1.0
      two y  2.0
foo one x  3.0
      two w  4.0

```

Other options in `set_index` allow you not drop the index columns or to add the index in-place (without creating a new object):

### 12.21.2 Reset the index

As a convenience, there is a new function on `DataFrame` called `reset_index()` which transfers the index values into the `DataFrame`'s columns and sets a simple integer index. This is the inverse operation of `set_index()`.

---

(continues on next page)

(continued from previous page)

```

0  bar  one  z  1.0
1  bar  two  y  2.0
2  foo  one  x  3.0
3  foo  two  w  4.0

```

The output is more similar to a SQL table or a record array. The names for the columns derived from the index are the ones stored in the `names` attribute.

You can use the `level` keyword to remove only a portion of the index:

```
In [337]: frame
```

```
Out [337]:
```

```

           c    d
c a    b
z bar one  z  1.0
y bar two  y  2.0
x foo one  x  3.0
w foo two  w  4.0

```

```
In [338]: frame.reset_index(level=1)
```

```

////////////////////////////////////
↪
           a    c    d
c b
z one  bar  z  1.0
y two  bar  y  2.0
x one  foo  x  3.0
w two  foo  w  4.0

```

`reset_index` takes an optional parameter `drop` which if true simply discards the index, instead of putting index values in the DataFrame's columns.

### 12.21.3 Adding an ad hoc index

If you create an index yourself, you can just assign it to the `index` field:

```
data.index = index
```

## 12.22 Returning a view versus a copy

When setting values in a pandas object, care must be taken to avoid what is called `chained indexing`. Here is an example.

```

In [339]: dfmi = pd.DataFrame([list('abcd'),
.....:                        list('efgh'),
.....:                        list('ijkl'),
.....:                        list('mnop')],
.....:                        columns=pd.MultiIndex.from_product([['one', 'two'],
.....:                                                              ['first', 'second']]),
↪ ']]))
.....:

In [340]: dfmi

```

(continues on next page)

(continued from previous page)

```
Out [340]:
      one      two
first second first second
0      a      b      c      d
1      e      f      g      h
2      i      j      k      l
3      m      n      o      p
```

Compare these two access methods:

```
In [341]: dfmi['one']['second']
Out [341]:
0      b
1      f
2      j
3      n
Name: second, dtype: object
```

```
In [342]: dfmi.loc[:, ('one', 'second')]
Out [342]:
0      b
1      f
2      j
3      n
Name: (one, second), dtype: object
```

These both yield the same results, so which should you use? It is instructive to understand the order of operations on these and why method 2 (`.loc`) is much preferred over method 1 (chained `[]`).

`dfmi['one']` selects the first level of the columns and returns a DataFrame that is singly-indexed. Then another Python operation `dfmi_with_one['second']` selects the series indexed by 'second'. This is indicated by the variable `dfmi_with_one` because pandas sees these operations as separate events. e.g. separate calls to `__getitem__`, so it has to treat them as linear operations, they happen one after another.

Contrast this to `df.loc[:, ('one', 'second')]` which passes a nested tuple of `(slice(None), ('one', 'second'))` to a single call to `__getitem__`. This allows pandas to deal with this as a single entity. Furthermore this order of operations *can* be significantly faster, and allows one to index *both* axes if so desired.

### 12.22.1 Why does assignment fail when using chained indexing?

The problem in the previous section is just a performance issue. What's up with the `SettingWithCopy` warning? We don't **usually** throw warnings around when you do something that might cost a few extra milliseconds!

But it turns out that assigning to the product of chained indexing has inherently unpredictable results. To see this, think about how the Python interpreter executes this code:

```
dfmi.loc[:, ('one', 'second')] = value
# becomes
dfmi.loc.__setitem__((slice(None), ('one', 'second')), value)
```

But this code is handled differently:

```
dfmi['one']['second'] = value
# becomes
dfmi.__getitem__('one').__setitem__('second', value)
```

See that `__getitem__` in there? Outside of simple cases, it's very hard to predict whether it will return a view or a copy (it depends on the memory layout of the array, about which pandas makes no guarantees), and therefore whether the `__setitem__` will modify `dfmi` or a temporary object that gets thrown out immediately afterward. **That's** what `SettingWithCopy` is warning you about!

---

**Note:** You may be wondering whether we should be concerned about the `loc` property in the first example. But `dfmi.loc` is guaranteed to be `dfmi` itself with modified indexing behavior, so `dfmi.loc.__getitem__` / `dfmi.loc.__setitem__` operate on `dfmi` directly. Of course, `dfmi.loc.__getitem__(idx)` may be a view or a copy of `dfmi`.

---

Sometimes a `SettingWithCopy` warning will arise at times when there's no obvious chained indexing going on. **These** are the bugs that `SettingWithCopy` is designed to catch! Pandas is probably trying to warn you that you've done this:

```
def do_something(df):
    foo = df[['bar', 'baz']] # Is foo a view? A copy? Nobody knows!
    # ... many lines here ...
    foo['quux'] = value      # We don't know whether this will modify df or not!
    return foo
```

Yikes!

## 12.22.2 Evaluation order matters

When you use chained indexing, the order and type of the indexing operation partially determine whether the result is a slice into the original object, or a copy of the slice.

Pandas has the `SettingWithCopyWarning` because assigning to a copy of a slice is frequently not intentional, but a mistake caused by chained indexing returning a copy where a slice was expected.

If you would like pandas to be more or less trusting about assignment to a chained indexing expression, you can set the `option.mode.chained_assignment` to one of these values:

- 'warn', the default, means a `SettingWithCopyWarning` is printed.
- 'raise' means pandas will raise a `SettingWithCopyException` you have to deal with.
- None will suppress the warnings entirely.

```
In [343]: dfb = pd.DataFrame({'a' : ['one', 'one', 'two',
.....:                             'three', 'two', 'one', 'six'],
.....:                       'c' : np.arange(7)})
.....:

# This will show the SettingWithCopyWarning
# but the frame values will be set
In [344]: dfb['c'][dfb.a.str.startswith('o')] = 42
```

This however is operating on a copy and will not work.

```
>>> pd.set_option('mode.chained_assignment', 'warn')
>>> dfb[dfb.a.str.startswith('o')]['c'] = 42
Traceback (most recent call last):
...
SettingWithCopyWarning:
```

(continues on next page)

(continued from previous page)

A value is trying to be set on a copy of a slice from a DataFrame.  
Try using `.loc[row_index,col_indexer] = value` instead

A chained assignment can also crop up in setting in a mixed dtype frame.

**Note:** These setting rules apply to all of `.loc/.iloc`.

This is the correct access method:

```
In [345]: dfc = pd.DataFrame({'A': ['aaa', 'bbb', 'ccc'], 'B': [1, 2, 3]})

In [346]: dfc.loc[0, 'A'] = 11

In [347]: dfc
Out[347]:
   A  B
0  11  1
1  bbb  2
2  ccc  3
```

This *can* work at times, but it is not guaranteed to, and therefore should be avoided:

```
In [348]: dfc = dfc.copy()

In [349]: dfc['A'][0] = 111

In [350]: dfc
Out[350]:
   A  B
0  111  1
1  bbb  2
2  ccc  3
```

This will **not** work at all, and so should be avoided:

```
>>> pd.set_option('mode.chained_assignment', 'raise')
>>> dfc.loc[0]['A'] = 1111
Traceback (most recent call last):
...
SettingWithCopyException:
  A value is trying to be set on a copy of a slice from a DataFrame.
  Try using .loc[row_index,col_indexer] = value instead
```

**Warning:** The chained assignment warnings / exceptions are aiming to inform the user of a possibly invalid assignment. There may be false positives; situations where a chained assignment is inadvertently reported.





## MULTIINDEX / ADVANCED INDEXING

This section covers indexing with a `MultiIndex` and more advanced indexing features.

See the [Indexing and Selecting Data](#) for general indexing documentation.

**Warning:** Whether a copy or a reference is returned for a setting operation, may depend on the context. This is sometimes called `chained assignment` and should be avoided. See [Returning a View versus Copy](#).

See the [cookbook](#) for some advanced strategies.

### 13.1 Hierarchical indexing (MultiIndex)

Hierarchical / Multi-level indexing is very exciting as it opens the door to some quite sophisticated data analysis and manipulation, especially for working with higher dimensional data. In essence, it enables you to store and manipulate data with an arbitrary number of dimensions in lower dimensional data structures like `Series` (1d) and `DataFrame` (2d).

In this section, we will show what exactly we mean by “hierarchical” indexing and how it integrates with all of the pandas indexing functionality described above and in prior sections. Later, when discussing [group by](#) and [pivoting and reshaping data](#), we’ll show non-trivial applications to illustrate how it aids in structuring data for analysis.

See the [cookbook](#) for some advanced strategies.

#### 13.1.1 Creating a MultiIndex (hierarchical index) object

The `MultiIndex` object is the hierarchical analogue of the standard `Index` object which typically stores the axis labels in pandas objects. You can think of `MultiIndex` as an array of tuples where each tuple is unique. A `MultiIndex` can be created from a list of arrays (using `MultiIndex.from_arrays`), an array of tuples (using `MultiIndex.from_tuples`), or a crossed set of iterables (using `MultiIndex.from_product`). The `Index` constructor will attempt to return a `MultiIndex` when it is passed a list of tuples. The following examples demonstrate different ways to initialize `MultiIndexes`.

```
In [1]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
...:             ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
...:

In [2]: tuples = list(zip(*arrays))

In [3]: tuples
Out[3]:
[('bar', 'one'),
```

(continues on next page)

(continued from previous page)

```

('bar', 'two'),
('baz', 'one'),
('baz', 'two'),
('foo', 'one'),
('foo', 'two'),
('qux', 'one'),
('qux', 'two')]

In [4]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [5]: index
Out[5]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
            labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
            names=['first', 'second'])

In [6]: s = pd.Series(np.random.randn(8), index=index)

In [7]: s
Out[7]:
first second
bar   one    0.469112
      two   -0.282863
baz   one   -1.509059
      two   -1.135632
foo   one    1.212112
      two   -0.173215
qux   one    0.119209
      two   -1.044236
dtype: float64

```

When you want every pairing of the elements in two iterables, it can be easier to use the `MultiIndex.from_product` function:

```

In [8]: iterables = [['bar', 'baz', 'foo', 'qux'], ['one', 'two']]

In [9]: pd.MultiIndex.from_product(iterables, names=['first', 'second'])
Out[9]:
MultiIndex(levels=[['bar', 'baz', 'foo', 'qux'], ['one', 'two']],
            labels=[[0, 0, 1, 1, 2, 2, 3, 3], [0, 1, 0, 1, 0, 1, 0, 1]],
            names=['first', 'second'])

```

As a convenience, you can pass a list of arrays directly into `Series` or `DataFrame` to construct a `MultiIndex` automatically:

```

In [10]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
.....:              np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]
.....:

In [11]: s = pd.Series(np.random.randn(8), index=arrays)

In [12]: s
Out[12]:
bar one    -0.861849
      two   -2.104569
baz one    -0.494929

```

(continues on next page)

(continued from previous page)

```

      two    1.071804
foo    one    0.721555
      two   -0.706771
qux    one   -1.039575
      two    0.271860
dtype: float64

```

```
In [13]: df = pd.DataFrame(np.random.randn(8, 4), index=arrays)
```

```
In [14]: df
```

Out [14] :

|     |     | 0         | 1         | 2         | 3         |
|-----|-----|-----------|-----------|-----------|-----------|
| bar | one | -0.424972 | 0.567020  | 0.276232  | -1.087401 |
|     | two | -0.673690 | 0.113648  | -1.478427 | 0.524988  |
| baz | one | 0.404705  | 0.577046  | -1.715002 | -1.039268 |
|     | two | -0.370647 | -1.157892 | -1.344312 | 0.844885  |
| foo | one | 1.075770  | -0.109050 | 1.643563  | -1.469388 |
|     | two | 0.357021  | -0.674600 | -1.776904 | -0.968914 |
| qux | one | -1.294524 | 0.413738  | 0.276662  | -0.472035 |
|     | two | -0.013960 | -0.362543 | -0.006154 | -0.923061 |

All of the `MultiIndex` constructors accept a `names` argument which stores string names for the levels themselves. If no names are provided, `None` will be assigned:

```
In [15]: df.index.names
```

```
Out[15]: FrozenList([None, None])
```

This index can back any axis of a pandas object, and the number of **levels** of the index is up to you:

```
In [16]: df = pd.DataFrame(np.random.randn(3, 8), index=['A', 'B', 'C'], columns=index)
```

```
In [17]: df
```

Out [17]:

| first  | bar       |          | baz       |           | foo       |           | qux       |           |
|--------|-----------|----------|-----------|-----------|-----------|-----------|-----------|-----------|
| second | one       | two      | one       | two       | one       | two       | one       | two       |
| A      | 0.895717  | 0.805244 | -1.206412 | 2.565646  | 1.431256  | 1.340309  | -1.170299 | -0.226169 |
| B      | 0.410835  | 0.813850 | 0.132003  | -0.827317 | -0.076467 | -1.187678 | 1.130127  | -1.436737 |
| C      | -1.413681 | 1.607920 | 1.024180  | 0.569605  | 0.875906  | -2.211372 | 0.974466  | -2.006747 |

```
In [18]: pd.DataFrame(np.random.randn(6, 6), index=index[:6], columns=index[:6])
```

|       |        | first second |           |           |           |           |           |
|-------|--------|--------------|-----------|-----------|-----------|-----------|-----------|
| first | second | bar one      | two       | baz one   | two       | foo one   | two       |
| bar   | one    | -0.410001    | -0.078638 | 0.545952  | -1.219217 | -1.226825 | 0.769804  |
|       | two    | -1.281247    | -0.727707 | -0.121306 | -0.097883 | 0.695775  | 0.341734  |
| baz   | one    | 0.959726     | -1.110336 | -0.619976 | 0.149748  | -0.732339 | 0.687738  |
|       | two    | 0.176444     | 0.403310  | -0.154951 | 0.301624  | -2.179861 | -1.369849 |
| foo   | one    | -0.954208    | 1.462696  | -1.743161 | -0.826591 | -0.345352 | 1.314232  |
|       | two    | 0.690579     | 0.995761  | 2.396780  | 0.014871  | 3.357427  | -0.317441 |

We've “sparsified” the higher levels of the indexes to make the console output a bit easier on the eyes. Note that how the index is displayed can be controlled using the `multi_sparse` option in `pandas.set_options()`:

```
In [19]: with pd.option_context('display.multi_sparse', False):
        ....:     df
        ....:
```

It's worth keeping in mind that there's nothing preventing you from using tuples as atomic labels on an axis:

```
In [20]: pd.Series(np.random.randn(8), index=tuples)
Out[20]:
(bar, one)    -1.236269
(bar, two)     0.896171
(baz, one)    -0.487602
(baz, two)    -0.082240
(foo, one)    -2.182937
(foo, two)     0.380396
(qux, one)     0.084844
(qux, two)     0.432390
dtype: float64
```

The reason that the `MultiIndex` matters is that it can allow you to do grouping, selection, and reshaping operations as we will describe below and in subsequent areas of the documentation. As you will see in later sections, you can find yourself working with hierarchically-indexed data without creating a `MultiIndex` explicitly yourself. However, when loading data from a file, you may wish to generate your own `MultiIndex` when preparing the data set.

### 13.1.2 Reconstructing the level labels

The method `get_level_values` will return a vector of the labels for each location at a particular level:

```
In [21]: index.get_level_values(0)
Out[21]: Index(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'], dtype='object',
↳ name='first')

In [22]: index.get_level_values('second')
////////////////////////////////////
↳ Index(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'], dtype='object',
↳ name='second')
```

### 13.1.3 Basic indexing on axis with MultiIndex

One of the important features of hierarchical indexing is that you can select data by a “partial” label identifying a subgroup in the data. **Partial** selection “drops” levels of the hierarchical index in the result in a completely analogous way to selecting a column in a regular `DataFrame`:

```
In [23]: df['bar']
Out[23]:
second      one      two
A      0.895717  0.805244
B      0.410835  0.813850
C     -1.413681  1.607920

In [24]: df['bar', 'one']
////////////////////////////////////
↳
A      0.895717
B      0.410835
```

(continues on next page)



```
In [31]: df[['foo', 'qux']].columns.remove_unused_levels()
Out[31]:
MultiIndex(levels=[['foo', 'qux'], ['one', 'two']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=['first', 'second'])
```

### 13.1.5 Data alignment and using `reindex`

Operations between differently-indexed objects having `MultiIndex` on the axes will work as you expect; data alignment will work the same as an `Index` of tuples:

```
In [32]: s + s[:-2]
Out[32]:
bar  one    -1.723698
      two    -4.209138
baz   one    -0.989859
      two     2.143608
foo   one     1.443110
      two    -1.413542
qux   one         NaN
      two         NaN
dtype: float64
```

```
In [33]: s + s[:,2]
```

```

////////////////////////////////////
↪
bar  one    -1.723698
      two         NaN
baz   one    -0.989859
      two         NaN
foo   one     1.443110
      two         NaN
qux   one    -2.079150
      two         NaN
dtype: float64
```

`reindex` can be called with another `MultiIndex`, or even a list or array of tuples:

```
In [34]: s.reindex(index[:3])
```

```
Out[34]:
first second
bar   one    -0.861849
      two    -2.104569
baz   one    -0.494929
dtype: float64
```

```
In [35]: s.reindex([('foo', 'two'), ('bar', 'one'), ('qux', 'one'), ('baz', 'one')])
```

```

////////////////////////////////////
↪
foo  two    -0.706771
bar  one    -0.861849
qux  one    -1.039575
baz  one    -0.494929
dtype: float64
```

## 13.2 Advanced indexing with hierarchical index

Syntactically integrating `MultiIndex` in advanced indexing with `.loc` is a bit challenging, but we’ve made every effort to do so. In general, `MultiIndex` keys take the form of tuples. For example, the following works as you would expect:

```
In [36]: df = df.T
```

```
In [37]: df
```

```
Out[37]:
```

|       |        | A         | B         | C         |
|-------|--------|-----------|-----------|-----------|
| first | second |           |           |           |
| bar   | one    | 0.895717  | 0.410835  | -1.413681 |
|       | two    | 0.805244  | 0.813850  | 1.607920  |
| baz   | one    | -1.206412 | 0.132003  | 1.024180  |
|       | two    | 2.565646  | -0.827317 | 0.569605  |
| foo   | one    | 1.431256  | -0.076467 | 0.875906  |
|       | two    | 1.340309  | -1.187678 | -2.211372 |
| qux   | one    | -1.170299 | 1.130127  | 0.974466  |
|       | two    | -0.226169 | -1.436737 | -2.006747 |

```
In [38]: df.loc[('bar', 'two'),]
```

```

////////////////////////////////////
↪
A    0.805244
B    0.813850
C    1.607920
Name: (bar, two), dtype: float64
```

Note that `df.loc['bar', 'two']` would also work in this example, but this shorthand notation can lead to ambiguity in general.

If you also want to index a specific column with `.loc`, you must use a tuple like this:

```
In [39]: df.loc[('bar', 'two'), 'A']
```

```
Out[39]: 0.80524402538637851
```

You don’t have to specify all levels of the `MultiIndex` by passing only the first elements of the tuple. For example, you can use “partial” indexing to get all elements with `bar` in the first level as follows:

```
df.loc['bar']
```

This is a shortcut for the slightly more verbose notation `df.loc[('bar',),]` (equivalent to `df.loc['bar',]` in this example).

“Partial” slicing also works quite nicely.

```
In [40]: df.loc['baz':'foo']
```

```
Out[40]:
```

|       |        | A         | B         | C         |
|-------|--------|-----------|-----------|-----------|
| first | second |           |           |           |
| baz   | one    | -1.206412 | 0.132003  | 1.024180  |
|       | two    | 2.565646  | -0.827317 | 0.569605  |
| foo   | one    | 1.431256  | -0.076467 | 0.875906  |
|       | two    | 1.340309  | -1.187678 | -2.211372 |

You can slice with a ‘range’ of values, by providing a slice of tuples.

```
In [41]: df.loc[('baz', 'two'):( 'qux', 'one')]
Out[41]:
```

|       |        | A         | B         | C         |
|-------|--------|-----------|-----------|-----------|
| first | second |           |           |           |
| baz   | two    | 2.565646  | -0.827317 | 0.569605  |
| foo   | one    | 1.431256  | -0.076467 | 0.875906  |
|       | two    | 1.340309  | -1.187678 | -2.211372 |
| qux   | one    | -1.170299 | 1.130127  | 0.974466  |

```
In [42]: df.loc[('baz', 'two'):'foo']
```

Passing a list of labels or tuples works similar to reindexing:

```
In [43]: df.loc[(['bar', 'two'), ('qux', 'one')]]
Out[43]:
```

|       |        | A         | B        | C        |
|-------|--------|-----------|----------|----------|
| first | second |           |          |          |
| bar   | two    | 0.805244  | 0.813850 | 1.607920 |
| qux   | one    | -1.170299 | 1.130127 | 0.974466 |

**Note:** It is important to note that tuples and lists are not treated identically in pandas when it comes to indexing. Whereas a tuple is interpreted as one multi-level key, a list is used to specify several keys. Or in other words, tuples go horizontally (traversing levels), lists go vertically (scanning levels).

Importantly, a list of tuples indexes several complete `MultiIndex` keys, whereas a tuple of lists refer to several values within a level:

```
In [44]: s = pd.Series([1, 2, 3, 4, 5, 6],
.....:                  index=pd.MultiIndex.from_product([["A", "B"], ["c", "d", "e",
↵"]]))
.....:
```

```
In [45]: s.loc[ [("A", "c"), ("B", "d")]] # list of tuples
```

```
Out [45]:
```

|   |   |   |
|---|---|---|
| A | c | 1 |
| B | d | 5 |

```
dtype: int64
```

```
In [46]: s.loc[(["A", "B"], ["c", "d"])] # tuple of lists
```

```
Out[46]:
A  c    1
   d    2
B  c    4
   d    5
dtype: int64
```



### 13.2.1 Using slicers

You can slice a `MultiIndex` by providing multiple indexers.

You can provide any of the selectors as if you are indexing by label, see [Selection by Label](#), including slices, lists of labels, labels, and boolean indexers.

You can use `slice(None)` to select all the contents of *that* level. You do not need to specify all the *deeper* levels, they will be implied as `slice(None)`.

As usual, **both sides** of the slicers are included as this is label indexing.

**Warning:** You should specify all axes in the `.loc` specifier, meaning the indexer for the **index** and for the **columns**. There are some ambiguous cases where the passed indexer could be mis-interpreted as indexing *both* axes, rather than into say the `MultiIndex` for the rows.

You should do this:

```
df.loc[(slice('A1', 'A3'), .....), :]
```

You should **not** do this:

```
df.loc[(slice('A1', 'A3'), .....)]
```

```
In [47]: def mklbl(prefix,n):
.....:     return ["%s%s" % (prefix,i) for i in range(n)]
.....:

In [48]: miindex = pd.MultiIndex.from_product([mklbl('A',4),
.....:                                         mklbl('B',2),
.....:                                         mklbl('C',4),
.....:                                         mklbl('D',2)])
.....:

In [49]: micolumns = pd.MultiIndex.from_tuples([('a','foo'),('a','bar'),
.....:                                         ('b','foo'),('b','bah')],
.....:                                         names=['lv10','lv11'])
.....:

In [50]: dfmi = pd.DataFrame(np.arange(len(miindex)*len(micolumns)).
↳ reshape((len(miindex),len(micolumns))),
.....:                               index=miindex,
.....:                               columns=micolumns).sort_index().sort_index(axis=1)
.....:

In [51]: dfmi
Out[51]:
lv10      a      b
lv11      bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
      D1    5    4    7    6
      C1 D0    9    8   11   10
      D1   13   12   15   14
      C2 D0   17   16   19   18
      D1   21   20   23   22
      C3 D0   25   24   27   26
...
A3 B1 C0 D1  229  228  231  230
```

(continues on next page)

(continued from previous page)

```

      C1 D0 233 232 235 234
        D1 237 236 239 238
      C2 D0 241 240 243 242
        D1 245 244 247 246
      C3 D0 249 248 251 250
        D1 253 252 255 254

[64 rows x 4 columns]

```

Basic multi-index slicing using slices, lists, and labels.

```

In [52]: dfmi.loc[(slice('A1','A3'), slice(None), ['C1', 'C3']), :]
Out[52]:
lvl0      a      b
lvl1    bar  foo  bah  foo
A1 B0 C1 D0   73   72   75   74
        D1   77   76   79   78
        C3 D0   89   88   91   90
        D1   93   92   95   94
      B1 C1 D0  105  104  107  106
        D1  109  108  111  110
        C3 D0  121  120  123  122
...
A3 B0 C1 D1  205  204  207  206
        C3 D0  217  216  219  218
        D1  221  220  223  222
      B1 C1 D0  233  232  235  234
        D1  237  236  239  238
        C3 D0  249  248  251  250
        D1  253  252  255  254

[24 rows x 4 columns]

```

You can use `pandas.IndexSlice` to facilitate a more natural syntax using `:`, rather than using `slice(None)`.

```

In [53]: idx = pd.IndexSlice

In [54]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
Out[54]:
lvl0      a      b
lvl1    foo  foo
A0 B0 C1 D0    8   10
        D1   12   14
        C3 D0   24   26
        D1   28   30
      B1 C1 D0   40   42
        D1   44   46
        C3 D0   56   58
...
A3 B0 C1 D1  204  206
        C3 D0  216  218
        D1  220  222
      B1 C1 D0  232  234
        D1  236  238
        C3 D0  248  250
        D1  252  254

```

(continues on next page)

(continued from previous page)

[32 rows x 2 columns]

It is possible to perform quite complicated selections using this method on multiple axes at the same time.

```
In [55]: dfmi.loc['A1', (slice(None), 'foo')]
```

```
Out[55]:
```

```
lvl0      a      b
lvl1      foo    foo
B0 C0 D0    64    66
      D1    68    70
      C1 D0    72    74
      D1    76    78
      C2 D0    80    82
      D1    84    86
      C3 D0    88    90
...
B1 C0 D1   100   102
      C1 D0   104   106
      D1   108   110
      C2 D0   112   114
      D1   116   118
      C3 D0   120   122
      D1   124   126
```

[16 rows x 2 columns]

```
In [56]: dfmi.loc[idx[:, :, ['C1', 'C3']], idx[:, 'foo']]
```

```
////////////////////////////////////
```

```
↪
lvl0      a      b
lvl1      foo    foo
A0 B0 C1 D0     8    10
      D1    12    14
      C3 D0    24    26
      D1    28    30
      B1 C1 D0    40    42
      D1    44    46
      C3 D0    56    58
...
A3 B0 C1 D1   204   206
      C3 D0   216   218
      D1   220   222
      B1 C1 D0   232   234
      D1   236   238
      C3 D0   248   250
      D1   252   254
```

[32 rows x 2 columns]

Using a boolean indexer you can provide selection related to the *values*.

```
In [57]: mask = dfmi[('a', 'foo')] > 200
```

```
In [58]: dfmi.loc[idx[mask, :, ['C1', 'C3']], idx[:, 'foo']]
```

```
Out[58]:
```

```
lvl0      a      b
lvl1      foo    foo
```

(continues on next page)

(continued from previous page)

```

A3 B0 C1 D1 204 206
      C3 D0 216 218
          D1 220 222
      B1 C1 D0 232 234
          D1 236 238
      C3 D0 248 250
          D1 252 254

```

You can also specify the `axis` argument to `.loc` to interpret the passed slicers on a single axis.

```
In [59]: dfmi.loc(axis=0)[: , :, ['C1', 'C3']]
```

```
Out[59]:
```

```

lvl0          a          b
lvl1      bar  foo  bah  foo
A0 B0 C1 D0    9    8   11   10
          D1   13   12   15   14
          C3 D0   25   24   27   26
          D1   29   28   31   30
      B1 C1 D0   41   40   43   42
          D1   45   44   47   46
          C3 D0   57   56   59   58
...
A3 B0 C1 D1  205  204  207  206
      C3 D0  217  216  219  218
          D1  221  220  223  222
      B1 C1 D0  233  232  235  234
          D1  237  236  239  238
          C3 D0  249  248  251  250
          D1  253  252  255  254

```

```
[32 rows x 4 columns]
```

Furthermore you can *set* the values using the following methods.

```
In [60]: df2 = dfmi.copy()
```

```
In [61]: df2.loc(axis=0)[: , :, ['C1', 'C3']] = -10
```

```
In [62]: df2
```

```
Out[62]:
```

```

lvl0          a          b
lvl1      bar  foo  bah  foo
A0 B0 C0 D0    1    0    3    2
          D1    5    4    7    6
          C1 D0  -10  -10  -10  -10
          D1  -10  -10  -10  -10
          C2 D0   17   16   19   18
          D1   21   20   23   22
          C3 D0  -10  -10  -10  -10
...
A3 B1 C0 D1  229  228  231  230
      C1 D0  -10  -10  -10  -10
          D1  -10  -10  -10  -10
          C2 D0  241  240  243  242
          D1  245  244  247  246
          C3 D0  -10  -10  -10  -10
          D1  -10  -10  -10  -10

```

(continues on next page)

(continued from previous page)

[64 rows x 4 columns]

You can use a right-hand-side of an alignable object as well.

```
In [63]: df2 = dfmi.copy()
```

```
In [64]: df2.loc[idx[:, :, ['C1', 'C3']], :] = df2 * 1000
```

```
In [65]: df2
```

```
Out[65]:
```

| lvl0 |    |    |    | a      | b      |        |        |
|------|----|----|----|--------|--------|--------|--------|
| lvl1 |    |    |    | bar    | foo    | bah    | foo    |
| A0   | B0 | C0 | D0 | 1      | 0      | 3      | 2      |
|      |    |    | D1 | 5      | 4      | 7      | 6      |
|      |    | C1 | D0 | 9000   | 8000   | 11000  | 10000  |
|      |    |    | D1 | 13000  | 12000  | 15000  | 14000  |
|      |    | C2 | D0 | 17     | 16     | 19     | 18     |
|      |    |    | D1 | 21     | 20     | 23     | 22     |
|      |    | C3 | D0 | 25000  | 24000  | 27000  | 26000  |
| ...  |    |    |    | ...    | ...    | ...    | ...    |
| A3   | B1 | C0 | D1 | 229    | 228    | 231    | 230    |
|      |    | C1 | D0 | 233000 | 232000 | 235000 | 234000 |
|      |    |    | D1 | 237000 | 236000 | 239000 | 238000 |
|      |    | C2 | D0 | 241    | 240    | 243    | 242    |
|      |    |    | D1 | 245    | 244    | 247    | 246    |
|      |    | C3 | D0 | 249000 | 248000 | 251000 | 250000 |
|      |    |    | D1 | 253000 | 252000 | 255000 | 254000 |

[64 rows x 4 columns]

## 13.2.2 Cross-section

The `xs` method of `DataFrame` additionally takes a `level` argument to make selecting data at a particular level of a `MultiIndex` easier.

```
In [66]: df
```

```
Out[66]:
```

|       |        | A         | B         | C         |
|-------|--------|-----------|-----------|-----------|
| first | second |           |           |           |
| bar   | one    | 0.895717  | 0.410835  | -1.413681 |
|       | two    | 0.805244  | 0.813850  | 1.607920  |
| baz   | one    | -1.206412 | 0.132003  | 1.024180  |
|       | two    | 2.565646  | -0.827317 | 0.569605  |
| foo   | one    | 1.431256  | -0.076467 | 0.875906  |
|       | two    | 1.340309  | -1.187678 | -2.211372 |
| qux   | one    | -1.170299 | 1.130127  | 0.974466  |
|       | two    | -0.226169 | -1.436737 | -2.006747 |

```
In [67]: df.xs('one', level='second')
```

```

////////////////////////////////////
↪
      A      B      C
first
bar  0.895717  0.410835 -1.413681
```

(continues on next page)

(continued from previous page)

```

baz    -1.206412    0.132003    1.024180
foo     1.431256   -0.076467    0.875906
qux    -1.170299    1.130127    0.974466

```

```

# using the slicers
In [68]: df.loc[(slice(None), 'one'), :]
Out [68]:

```

|       |        | A         | B         | C         |
|-------|--------|-----------|-----------|-----------|
| first | second |           |           |           |
| bar   | one    | 0.895717  | 0.410835  | -1.413681 |
| baz   | one    | -1.206412 | 0.132003  | 1.024180  |
| foo   | one    | 1.431256  | -0.076467 | 0.875906  |
| qux   | one    | -1.170299 | 1.130127  | 0.974466  |

You can also select on the columns with `xs()`, by providing the axis argument.

```

In [69]: df = df.T

In [70]: df.xs('one', level='second', axis=1)
Out [70]:

```

|   | first     | bar       | baz       | foo       | qux |
|---|-----------|-----------|-----------|-----------|-----|
| A | 0.895717  | -1.206412 | 1.431256  | -1.170299 |     |
| B | 0.410835  | 0.132003  | -0.076467 | 1.130127  |     |
| C | -1.413681 | 1.024180  | 0.875906  | 0.974466  |     |

```

# using the slicers
In [71]: df.loc[:, (slice(None), 'one')]
Out [71]:

```

|        | first     | bar       | baz       | foo       | qux |
|--------|-----------|-----------|-----------|-----------|-----|
| second | one       | one       | one       | one       | one |
| A      | 0.895717  | -1.206412 | 1.431256  | -1.170299 |     |
| B      | 0.410835  | 0.132003  | -0.076467 | 1.130127  |     |
| C      | -1.413681 | 1.024180  | 0.875906  | 0.974466  |     |

`xs()` also allows selection with multiple keys.

```

In [72]: df.xs(('one', 'bar'), level=('second', 'first'), axis=1)
Out [72]:

```

|        | first     | bar |
|--------|-----------|-----|
| second | one       |     |
| A      | 0.895717  |     |
| B      | 0.410835  |     |
| C      | -1.413681 |     |

```

# using the slicers
In [73]: df.loc[:, ('bar', 'one')]
Out [73]:
A    0.895717
B    0.410835
C   -1.413681
Name: (bar, one), dtype: float64

```

You can pass `drop_level=False` to `xs()` to retain the level that was selected.

```
In [74]: df.xs('one', level='second', axis=1, drop_level=False)
Out[74]:
```

|        | first     | bar       | baz       | foo       | qux |
|--------|-----------|-----------|-----------|-----------|-----|
| second | one       | one       | one       | one       | one |
| A      | 0.895717  | -1.206412 | 1.431256  | -1.170299 |     |
| B      | 0.410835  | 0.132003  | -0.076467 | 1.130127  |     |
| C      | -1.413681 | 1.024180  | 0.875906  | 0.974466  |     |

Compare the above with the result using `drop_level=True` (the default value).

```
In [75]: df.xs('one', level='second', axis=1, drop_level=True)
Out[75]:
```

|   | first     | bar       | baz       | foo       | qux |
|---|-----------|-----------|-----------|-----------|-----|
| A | 0.895717  | -1.206412 | 1.431256  | -1.170299 |     |
| B | 0.410835  | 0.132003  | -0.076467 | 1.130127  |     |
| C | -1.413681 | 1.024180  | 0.875906  | 0.974466  |     |

### 13.2.3 Advanced reindexing and alignment

The parameter `level` has been added to the `reindex` and `align` methods of pandas objects. This is useful to broadcast values across a level. For instance:

[illegible]

```
In [77]: df = pd.DataFrame(np.random.randn(4,2), index=midx)
```

```
In [78]: df
```

```
Out [78]:
```

|      |   | 0        | 1         |
|------|---|----------|-----------|
| one  | y | 1.519970 | -0.493662 |
|      | x | 0.600178 | 0.274230  |
| zero | y | 0.132885 | -0.023688 |
|      | x | 2.410179 | 1.450520  |

```
In [79]: df2 = df.mean(level=0)
```

```
In [80]: df2
```

```
Out[80]:
```

|      | 0        | 1         |
|------|----------|-----------|
| one  | 1.060074 | -0.109716 |
| zero | 1.271532 | 0.713416  |

```
In [81]: df2.reindex(df.index, level=0)
```

Out

|      |   | 0        | 1         |
|------|---|----------|-----------|
| one  | y | 1.060074 | -0.109716 |
|      | x | 1.060074 | -0.109716 |
| zero | y | 1.271532 | 0.713416  |
|      | x | 1.271532 | 0.713416  |

```
# aligning
```

```
In [82]: df_aligned, df2_aligned = df.align(df2, level=0)
```

(continues on next page)

(continued from previous page)

```
In [83]: df_aligned
Out[83]:
```

|      |   | 0        | 1         |
|------|---|----------|-----------|
| one  | y | 1.519970 | -0.493662 |
|      | x | 0.600178 | 0.274230  |
| zero | y | 0.132885 | -0.023688 |
|      | x | 2.410179 | 1.450520  |

```
In [84]: df2_aligned
////////////////////////////////////
```

|      |   | 0        | 1         |
|------|---|----------|-----------|
| one  | y | 1.060074 | -0.109716 |
|      | x | 1.060074 | -0.109716 |
| zero | y | 1.271532 | 0.713416  |
|      | x | 1.271532 | 0.713416  |

### 13.2.4 Swapping levels with `swaplevel()`

The `swaplevel` function can switch the order of two levels:

```
In [85]: df[:5]
Out[85]:
```

|      |   | 0        | 1         |
|------|---|----------|-----------|
| one  | y | 1.519970 | -0.493662 |
|      | x | 0.600178 | 0.274230  |
| zero | y | 0.132885 | -0.023688 |
|      | x | 2.410179 | 1.450520  |

```
In [86]: df[:5].swaplevel(0, 1, axis=0)
////////////////////////////////////
```

|        |  | 0        | 1         |
|--------|--|----------|-----------|
| y one  |  | 1.519970 | -0.493662 |
| x one  |  | 0.600178 | 0.274230  |
| y zero |  | 0.132885 | -0.023688 |
| x zero |  | 2.410179 | 1.450520  |

### 13.2.5 Reordering levels with `reorder_levels()`

The `reorder_levels` function generalizes the `swaplevel` function, allowing you to permute the hierarchical index levels in one step:

```
In [87]: df[:5].reorder_levels([1,0], axis=0)
Out[87]:
```

|        |  | 0        | 1         |
|--------|--|----------|-----------|
| y one  |  | 1.519970 | -0.493662 |
| x one  |  | 0.600178 | 0.274230  |
| y zero |  | 0.132885 | -0.023688 |
| x zero |  | 2.410179 | 1.450520  |



## 13.3 Sorting a MultiIndex

For MultiIndex-ed objects to be indexed and sliced effectively, they need to be sorted. As with any index, you can use `sort_index`.

```
In [88]: import random; random.shuffle(tuples)

In [89]: s = pd.Series(np.random.randn(8), index=pd.MultiIndex.from_tuples(tuples))

In [90]: s
Out[90]:
foo one    0.206053
qux one   -0.251905
baz two   -2.213588
bar two    1.063327
qux two    1.266143
foo two    0.299368
baz one   -0.863838
bar one    0.408204
dtype: float64

In [91]: s.sort_index()
//////////
↪
bar one    0.408204
   two    1.063327
baz one   -0.863838
   two   -2.213588
foo one    0.206053
   two    0.299368
qux one   -0.251905
   two    1.266143
dtype: float64

In [92]: s.sort_index(level=0)
//////////
↪
bar one    0.408204
   two    1.063327
baz one   -0.863838
   two   -2.213588
foo one    0.206053
   two    0.299368
qux one   -0.251905
   two    1.266143
dtype: float64

In [93]: s.sort_index(level=1)
//////////
↪
bar one    0.408204
baz one   -0.863838
foo one    0.206053
qux one   -0.251905
bar two    1.063327
baz two   -2.213588
foo two    0.299368
```

(continues on next page)

(continued from previous page)

```
qux two 1.266143
dtype: float64
```

You may also pass a level name to `sort_index` if the `MultiIndex` levels are named.

```
In [94]: s.index.set_names(['L1', 'L2'], inplace=True)
```

```
In [95]: s.sort_index(level='L1')
```

```
Out [95]:
```

```
L1 L2
bar one 0.408204
      two 1.063327
baz one -0.863838
      two -2.213588
foo one 0.206053
      two 0.299368
qux one -0.251905
      two 1.266143
dtype: float64
```

```
In [96]: s.sort_index(level='L2')
```

```

////////////////////////////////////
↪
L1 L2
bar one 0.408204
baz one -0.863838
foo one 0.206053
qux one -0.251905
bar two 1.063327
baz two -2.213588
foo two 0.299368
qux two 1.266143
dtype: float64
```

On higher dimensional objects, you can sort any of the other axes by level if they have a `MultiIndex`:

```
In [97]: dfm.T.sort_index(level=1, axis=1)
```

```
Out [97]:
```

```

      one      zero      one      zero
      x      x      y      y
0  0.600178  2.410179  1.519970  0.132885
1  0.274230  1.450520 -0.493662 -0.023688
```

Indexing will work even if the data are not sorted, but will be rather inefficient (and show a `PerformanceWarning`). It will also return a copy of the data rather than a view:

```
In [98]: dfm = pd.DataFrame({'jim': [0, 0, 1, 1],
.....:                      'joe': ['x', 'x', 'z', 'y'],
.....:                      'jolie': np.random.rand(4)})
.....:
```

```
In [99]: dfm = dfm.set_index(['jim', 'joe'])
```

```
In [100]: dfm
```

```
Out [100]:
      jolie
```

(continues on next page)

```
jim joe
0  x  0.490671
   x  0.120248
1  z  0.537020
   y  0.110968
```

## 13.4 Take Methods

Similar to NumPy ndarrays, pandas Index, Series, and DataFrame also provides the `take` method that retrieves elements along a given axis at the given indices. The given indices must be either a list or an ndarray of integer index positions. `take` will also accept negative integers as relative positions to the end of the object.

```
In [108]: index = pd.Index(np.random.randint(0, 1000, 10))

In [109]: index
Out[109]: Int64Index([214, 502, 712, 567, 786, 175, 993, 133, 758, 329], dtype='int64',
↳)

In [110]: positions = [0, 9, 3]

In [111]: index[positions]
Out[111]: Int64Index([214, 329, 567], dtype='int64')

In [112]: index.take(positions)
\\Out[112]: Int64Index([214, 329,
↳567], dtype='int64')

In [113]: ser = pd.Series(np.random.randn(10))

In [114]: ser.iloc[positions]
Out[114]:
0    -0.179666
9     1.824375
3     0.392149
dtype: float64

In [115]: ser.take(positions)
\\Out[115]:
0    -0.179666
9     1.824375
3     0.392149
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```
In [116]: frm = pd.DataFrame(np.random.randn(5, 3))

In [117]: frm.take([1, 4, 3])
Out[117]:
      0         1         2
1 -1.237881  0.106854 -1.276829
4  0.629675 -1.425966  1.857704
3  0.979542 -1.633678  0.615855

In [118]: frm.take([0, 2], axis=1)
\\
↳
      0         2
0  0.595974  0.601544
1 -1.237881 -1.276829
2 -0.767101  1.499591
3  0.979542  0.615855
4  0.629675  1.857704
```

It is important to note that the `take` method on pandas objects are not intended to work on boolean indices and may return unexpected results.

```
In [119]: arr = np.random.randn(10)

In [120]: arr.take([False, False, True, True])
Out[120]: array([-1.1935, -1.1935,  0.6775,  0.6775])

In [121]: arr[[0, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[121]: array([-1.1935,  0.
↪6775])

In [122]: ser = pd.Series(np.random.randn(10))

In [123]: ser.take([False, False, True, True])
Out[123]:
0    0.233141
0    0.233141
1   -0.223540
1   -0.223540
dtype: float64

In [124]: ser.iloc[[0, 1]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[124]: ↵
↪
0    0.233141
1   -0.223540
dtype: float64
```

Finally, as a small note on performance, because the `take` method handles a narrower range of inputs, it can offer performance that is a good deal faster than fancy indexing.

## 13.5 Index Types

We have discussed `MultiIndex` in the previous sections pretty extensively. `DatetimeIndex` and `PeriodIndex` are shown [here](#), and information about `TimedeltaIndex` is found [here](#).

In the following sub-sections we will highlight some other index types.

### 13.5.1 CategoricalIndex

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a `Categorical` and allows efficient indexing and storage of an index with a large number of duplicated elements.

```
In [125]: from pandas.api.types import CategoricalDtype

In [126]: df = pd.DataFrame({'A': np.arange(6),
.....:                      'B': list('aabbca')})
.....:

In [127]: df['B'] = df['B'].astype(CategoricalDtype(list('cab')))

In [128]: df
```

---

(continues on next page)

(continued from previous page)

```

Out[128]:
   A  B
0  0  a
1  1  a
2  2  b
3  3  b
4  4  c
5  5  a

In [129]: df.dtypes
Out[129]:
A      int64
B      category
dtype: object

In [130]: df.B.cat.categories
Out[130]:
Index(['c', 'a', 'b'], dtype='object')

```

Setting the index will create a CategoricalIndex.

```

In [131]: df2 = df.set_index('B')

In [132]: df2.index
Out[132]: CategoricalIndex(['a', 'a', 'b', 'b', 'c', 'a'], categories=['c', 'a', 'b'],
↳ ordered=False, name='B', dtype='category')

```

Indexing with `__getitem__`/`.iloc`/`.loc` works similarly to an Index with duplicates. The indexers **must** be in the category or the operation will raise a `KeyError`.

```

In [133]: df2.loc['a']
Out[133]:
A
B
a  0
a  1
a  5

```

The CategoricalIndex is **preserved** after indexing:

```

In [134]: df2.loc['a'].index
Out[134]: CategoricalIndex(['a', 'a', 'a'], categories=['c', 'a', 'b'], ordered=False,
↳ name='B', dtype='category')

```

Sorting the index will sort by the order of the categories (recall that we created the index with `CategoricalDtype(list('cab'))`, so the sorted order is cab).

```

In [135]: df2.sort_index()
Out[135]:
A
B
c  4
a  0
a  1
a  5
b  2
b  3

```

Groupby operations on the index will preserve the index nature as well.

```
In [136]: df2.groupby(level=0).sum()
Out[136]:
      A
B
c    4
a    6
b    5

In [137]: df2.groupby(level=0).sum().index
Out[137]: CategoricalIndex(['c', 'a', 'b'],
categories=['c', 'a', 'b'], ordered=False, name='B', dtype='category')
```

Reindexing operations will return a resulting index based on the type of the passed indexer. Passing a list will return a plain-old Index; indexing with a Categorical will return a CategoricalIndex, indexed according to the categories of the **passed** Categorical dtype. This allows one to arbitrarily index these even with values **not** in the categories, similarly to how you can reindex **any** pandas index.

```
In [138]: df2.reindex(['a', 'e'])
Out[138]:
      A
B
a    0.0
a    1.0
a    5.0
e    NaN

In [139]: df2.reindex(['a', 'e']).index
Out[139]: Index(['a', 'a', 'a', 'e'], dtype='object', name='B')

In [140]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde')))
Out[140]:
      A
B
a    0.0
a    1.0
a    5.0
e    NaN

In [141]: df2.reindex(pd.Categorical(['a', 'e'], categories=list('abcde'))).index
Out[141]: CategoricalIndex(['a', 'a', 'a', 'e'], categories=['a', 'b', 'c', 'd', 'e'],
ordered=False, name='B', dtype='category')
```

**Warning:** Reshaping and Comparison operations on a CategoricalIndex must have the same categories or a TypeError will be raised.

```
In [9]: df3 = pd.DataFrame({'A' : np.arange(6),
                           'B' : pd.Series(list('aabbca')).astype('category')})

In [11]: df3 = df3.set_index('B')

In [11]: df3.index
Out[11]: CategoricalIndex([u'a', u'a', u'b', u'b', u'c', u'a'], categories=[u'a', u'b', u'c'], ordered=False, name='B', dtype='category')
```

```
In [12]: pd.concat([df2, df3])
```

## 13.5.2 Int64Index and RangeIndex

**Warning:** Indexing on an integer-based Index with floats has been clarified in 0.18.0, for a summary of the changes, see [here](#).

`Int64Index` is a fundamental basic index in pandas. This is an Immutable array implementing an ordered, sliceable set. Prior to 0.18.0, the `Int64Index` would provide the default index for all `NDFrame` objects.

`RangeIndex` is a sub-class of `Int64Index` added in version 0.18.0, now providing the default index for all `NDFrame` objects. `RangeIndex` is an optimized version of `Int64Index` that can represent a monotonic ordered set. These are analogous to Python [range types](#).

## 13.5.3 Float64Index

By default a `Float64Index` will be automatically created when passing floating, or mixed-integer-floating values in index creation. This enables a pure label-based slicing paradigm that makes `[]`, `ix`, `loc` for scalar indexing and slicing work exactly the same.

```
In [142]: indexf = pd.Index([1.5, 2, 3, 4.5, 5])

In [143]: indexf
Out[143]: Float64Index([1.5, 2.0, 3.0, 4.5, 5.0], dtype='float64')

In [144]: sf = pd.Series(range(5), index=indexf)

In [145]: sf
Out[145]:
1.5    0
2.0    1
3.0    2
4.5    3
5.0    4
dtype: int64
```

Scalar selection for `[]`, `.loc` will always be label based. An integer will match an equal float index (e.g. 3 is equivalent to 3.0).

```
In [146]: sf[3]
Out[146]: 2

In [147]: sf[3.0]
Out[147]: 2

In [148]: sf.loc[3]
Out[148]: 2

In [149]: sf.loc[3.0]
Out[149]: 2
```

The only positional indexing is via `iloc`.



```
In [150]: sf.iloc[3]
Out[150]: 3
```

A scalar index that is not found will raise a `KeyError`. Slicing is primarily on the values of the index when using `[]`, `ix`, `loc`, and **always** positional when using `iloc`. The exception is when the slice is boolean, in which case it will always be positional.

```
In [151]: sf[2:4]
Out[151]:
2.0    1
3.0    2
dtype: int64

In [152]: sf.loc[2:4]
Out[152]:
2.0    1
3.0    2
dtype: int64

In [153]: sf.iloc[2:4]
Out[153]:
3.0    2
4.5    3
dtype: int64
```

In float indexes, slicing using floats is allowed.

```
In [154]: sf[2.1:4.6]
Out[154]:
3.0    2
4.5    3
dtype: int64

In [155]: sf.loc[2.1:4.6]
Out[155]:
3.0    2
4.5    3
dtype: int64
```

In non-float indexes, slicing using floats will raise a `TypeError`.

```
In [1]: pd.Series(range(5))[3.5]
TypeError: the label [3.5] is not a proper indexer for this index type (Int64Index)

In [1]: pd.Series(range(5))[3.5:4.5]
TypeError: the slice start [3.5] is not a proper indexer for this index type
(Int64Index)
```

**Warning:** Using a scalar float indexer for `.iloc` has been removed in 0.18.0, so the following will raise a `TypeError`:

```
In [3]: pd.Series(range(5)).iloc[3.0]
TypeError: cannot do positional indexing on <class 'pandas.indexes.range.RangeIndex'> with these indexers [3.0] of <type 'float'>
```

Here is a typical use-case for using this type of indexing. Imagine that you have a somewhat irregular timedelta-like indexing scheme, but the data is recorded as floats. This could for example be millisecond offsets.

```
In [156]: dfir = pd.concat([pd.DataFrame(np.random.randn(5,2),
.....:                                index=np.arange(5) * 250.0,
.....:                                columns=list('AB')),
.....:                    pd.DataFrame(np.random.randn(6,2),
.....:                                index=np.arange(4,10) * 250.1,
.....:                                columns=list('AB'))])
```

```
In [157]: dfir
```

Out [157]:

|        | A         | B         |
|--------|-----------|-----------|
| 0.0    | 0.997289  | -1.693316 |
| 250.0  | -0.179129 | -1.598062 |
| 500.0  | 0.936914  | 0.912560  |
| 750.0  | -1.003401 | 1.632781  |
| 1000.0 | -0.724626 | 0.178219  |
| 1000.4 | 0.310610  | -0.108002 |
| 1250.5 | -0.974226 | -1.147708 |
| 1500.6 | -2.281374 | 0.760010  |
| 1750.7 | -0.742532 | 1.533318  |
| 2000.8 | 2.495362  | -0.432771 |
| 2250.9 | -0.068954 | 0.043520  |

Selection operations then will always work on a value basis, for all selection operators.

```
In [158]: dfir[0:1000.4]
```

Out [158] :

|        | A         | B         |
|--------|-----------|-----------|
| 0.0    | 0.997289  | -1.693316 |
| 250.0  | -0.179129 | -1.598062 |
| 500.0  | 0.936914  | 0.912560  |
| 750.0  | -1.003401 | 1.632781  |
| 1000.0 | -0.724626 | 0.178219  |
| 1000.4 | 0.310610  | -0.108002 |

```
In [159]: dfir.loc[0:1001, 'A']
```

```
↔
0.0      0.997289
250.0    -0.179129
500.0     0.936914
750.0    -1.003401
1000.0   -0.724626
1000.4    0.310610
Name: A, dtype: float64
```

```
In [160]: dfir.loc[1000.4]
```

```
↪
A      0.310610
B     -0.108002
Name: 1000.4, dtype: float64
```

You could retrieve the first 1 second (1000 ms) of data as such:

```
In [161]: dfir[0:1000]
Out[161]:
```

|        | A         | B         |
|--------|-----------|-----------|
| 0.0    | 0.997289  | -1.693316 |
| 250.0  | -0.179129 | -1.598062 |
| 500.0  | 0.936914  | 0.912560  |
| 750.0  | -1.003401 | 1.632781  |
| 1000.0 | -0.724626 | 0.178219  |

If you need integer based selection, you should use `iloc`:

```
In [162]: dfir.iloc[0:5]
Out[162]:
```

|        | A         | B         |
|--------|-----------|-----------|
| 0.0    | 0.997289  | -1.693316 |
| 250.0  | -0.179129 | -1.598062 |
| 500.0  | 0.936914  | 0.912560  |
| 750.0  | -1.003401 | 1.632781  |
| 1000.0 | -0.724626 | 0.178219  |

### 13.5.4 IntervalIndex

New in version 0.20.0.

*IntervalIndex* together with its own dtype, `interval` as well as the *Interval* scalar type, allow first-class support in pandas for interval notation.

The `IntervalIndex` allows some unique indexing and is also used as a return type for the categories in `cut()` and `qcut()`.

**Warning:** These indexing behaviors are provisional and may change in a future version of pandas.

An `IntervalIndex` can be used in `Series` and in `DataFrame` as the index.

```
In [163]: df = pd.DataFrame({'A': [1, 2, 3, 4]},
.....:                      index=pd.IntervalIndex.from_breaks([0, 1, 2, 3, 4]))
.....:

In [164]: df
Out[164]:
```

|        | A |
|--------|---|
| (0, 1] | 1 |
| (1, 2] | 2 |
| (2, 3] | 3 |
| (3, 4] | 4 |

Label based indexing via `.loc` along the edges of an interval works as you would expect, selecting that particular interval.

```
In [165]: df.loc[2]
Out[165]:
```

| A |
|---|
| 2 |

Name: (1, 2], dtype: int64

(continues on next page)

(continued from previous page)

```
In [166]: df.loc[[2, 3]]
Out[166]:
      A
(1, 2] 2
(2, 3] 3
```

If you select a label *contained* within an interval, this will also select the interval.

```
In [167]: df.loc[2.5]
Out[167]:
A      3
Name: (2, 3], dtype: int64

In [168]: df.loc[[2.5, 3.5]]
Out[168]:
      A
(2, 3] 3
(3, 4] 4
```

Interval and IntervalIndex are used by cut and qcut:

```
In [169]: c = pd.cut(range(4), bins=2)

In [170]: c
Out[170]:
[(-0.003, 1.5], (-0.003, 1.5], (1.5, 3.0], (1.5, 3.0]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]

In [171]: c.categories
Out[171]:
IntervalIndex([(-0.003, 1.5], (1.5, 3.0]]
              closed='right',
              dtype='interval[float64]')
```

Furthermore, IntervalIndex allows one to bin *other* data with these same bins, with NaN representing a missing value similar to other dtypes.

```
In [172]: pd.cut([0, 3, 5, 1], bins=c.categories)
Out[172]:
[(-0.003, 1.5], (1.5, 3.0], NaN, (-0.003, 1.5]]
Categories (2, interval[float64]): [(-0.003, 1.5] < (1.5, 3.0]]
```

### 13.5.4.1 Generating Ranges of Intervals

If we need intervals on a regular frequency, we can use the `interval_range()` function to create an IntervalIndex using various combinations of start, end, and periods. The default frequency for `interval_range` is a 1 for numeric intervals, and calendar day for datetime-like intervals:

```
In [173]: pd.interval_range(start=0, end=5)
Out[173]:
IntervalIndex([ (0, 1], (1, 2], (2, 3], (3, 4], (4, 5]]
              closed='right',
              dtype='interval[int64]')
```

(continues on next page)

(continued from previous page)

```

In [174]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4)
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03], (2017-01-03, 2017-
01-04], (2017-01-04, 2017-01-05]]
           closed='right',
           dtype='interval[datetime64[ns]]')

In [175]: pd.interval_range(end=pd.Timedelta('3 days'), periods=3)
IntervalIndex([(0 days 00:00:00, 1 days 00:00:00], (1 days 00:00:00, 2 days 00:00:00],
(2 days 00:00:00, 3 days 00:00:00]]
           closed='right',
           dtype='interval[timedelta64[ns]]')

```

The `freq` parameter can be used to specify non-default frequencies, and can utilize a variety of *frequency aliases* with datetime-like intervals:

```

In [176]: pd.interval_range(start=0, periods=5, freq=1.5)
Out[176]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0], (6.0, 7.5]]
           closed='right',
           dtype='interval[float64]')

In [177]: pd.interval_range(start=pd.Timestamp('2017-01-01'), periods=4, freq='W')
IntervalIndex([(2017-01-01, 2017-01-08], (2017-01-08, 2017-01-15], (2017-01-15, 2017-
01-22], (2017-01-22, 2017-01-29]]
           closed='right',
           dtype='interval[datetime64[ns]]')

In [178]: pd.interval_range(start=pd.Timedelta('0 days'), periods=3, freq='9H')
IntervalIndex([(0 days 00:00:00, 0 days 09:00:00], (0 days 09:00:00, 0 days 18:00:00],
(0 days 18:00:00, 1 days 03:00:00]]
           closed='right',
           dtype='interval[timedelta64[ns]]')

```

Additionally, the `closed` parameter can be used to specify which side(s) the intervals are closed on. Intervals are closed on the right side by default.

```

In [179]: pd.interval_range(start=0, end=4, closed='both')
Out[179]:
IntervalIndex([[0, 1], [1, 2], [2, 3], [3, 4]]
           closed='both',
           dtype='interval[int64]')

In [180]: pd.interval_range(start=0, end=4, closed='neither')
IntervalIndex([(0, 1), (1, 2), (2, 3), (3, 4)]
           closed='neither',
           dtype='interval[int64]')

```

New in version 0.23.0.

Specifying start, end, and periods will generate a range of evenly spaced intervals from start to end inclusively, with periods number of elements in the resulting IntervalIndex:

```
In [181]: pd.interval_range(start=0, end=6, periods=4)
Out[181]:
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]]
              closed='right',
              dtype='interval[float64]')

In [182]: pd.interval_range(pd.Timestamp('2018-01-01'), pd.Timestamp('2018-02-28'),
↪periods=3)
↪
IntervalIndex([(2018-01-01, 2018-01-20 08:00:00], (2018-01-20 08:00:00, 2018-02-08
↪16:00:00], (2018-02-08 16:00:00, 2018-02-28]]
              closed='right',
              dtype='interval[datetime64[ns]]')
```

## 13.6 Miscellaneous indexing FAQ

### 13.6.1 Integer indexing

Label-based indexing with integer axis labels is a thorny topic. It has been discussed heavily on mailing lists and among various members of the scientific Python community. In pandas, our general viewpoint is that labels matter more than integer locations. Therefore, with an integer axis index *only* label-based indexing is possible with the standard tools like `.loc`. The following code will generate exceptions:

```
s = pd.Series(range(5))
s[-1]
df = pd.DataFrame(np.random.randn(5, 4))
df
df.loc[-2:]
```

This deliberate decision was made to prevent ambiguities and subtle bugs (many users reported finding bugs when the API change was made to stop “falling back” on position-based indexing).

### 13.6.2 Non-monotonic indexes require exact matches

If the index of a Series or DataFrame is monotonically increasing or decreasing, then the bounds of a label-based slice can be outside the range of the index, much like slice indexing a normal Python list. Monotonicity of an index can be tested with the `is_monotonic_increasing` and `is_monotonic_decreasing` attributes.

```
In [183]: df = pd.DataFrame(index=[2,3,3,4,5], columns=['data'], data=list(range(5)))

In [184]: df.index.is_monotonic_increasing
Out[184]: True

# no rows 0 or 1, but still returns rows 2, 3 (both of them), and 4:
In [185]: df.loc[0:4, :]
Out[185]:
data
```

(continues on next page)

(continued from previous page)

```

2      0
3      1
3      2
4      3

# slice is are outside the index, so empty DataFrame is returned
In [186]: df.loc[13:15, :]
Out[186]:
Empty DataFrame
Columns: [data]
Index: []

```

On the other hand, if the index is not monotonic, then both slice bounds must be *unique* members of the index.

```

In [187]: df = pd.DataFrame(index=[2,3,1,4,3,5], columns=['data'],
↳data=list(range(6)))

In [188]: df.index.is_monotonic_increasing
Out[188]: False

# OK because 2 and 4 are in the index
In [189]: df.loc[2:4, :]
Out[189]:
   data
2     0
3     1
1     2
4     3

```

```

# 0 is not in the index
In [9]: df.loc[0:4, :]
KeyError: 0

# 3 is not a unique label
In [11]: df.loc[2:3, :]
KeyError: 'Cannot get right slice bound for non-unique label: 3'

```

`Index.is_monotonic_increasing()` and `Index.is_monotonic_decreasing()` only check that an index is weakly monotonic. To check for strict monotonicity, you can combine one of those with `Index.is_unique()`

```

In [190]: weakly_monotonic = pd.Index(['a', 'b', 'c', 'c'])

In [191]: weakly_monotonic
Out[191]: Index(['a', 'b', 'c', 'c'], dtype='object')

In [192]: weakly_monotonic.is_monotonic_increasing
Out[192]: True

In [193]: weakly_monotonic.is_monotonic_increasing & weakly_monotonic.is_unique
Out[193]: False

```

### 13.6.3 Endpoints are inclusive

Compared with standard Python sequence slicing in which the slice endpoint is not inclusive, label-based slicing in pandas **is inclusive**. The primary reason for this is that it is often not possible to easily determine the “successor” or next element after a particular label in an index. For example, consider the following Series:

```
In [194]: s = pd.Series(np.random.randn(6), index=list('abcdef'))

In [195]: s
Out[195]:
a    0.112246
b    0.871721
c   -0.816064
d   -0.784880
e    1.030659
f    0.187483
dtype: float64
```

Suppose we wished to slice from c to e, using integers this would be accomplished as such:

```
In [196]: s[2:5]
Out[196]:
c   -0.816064
d   -0.784880
e    1.030659
dtype: float64
```

However, if you only had c and e, determining the next element in the index can be somewhat complicated. For example, the following does not work:

```
s.loc['c':'e'+1]
```

A very common use case is to limit a time series to start and end at two specific dates. To enable this, we made the design to make label-based slicing include both endpoints:

```
In [197]: s.loc['c':'e']
Out[197]:
c   -0.816064
d   -0.784880
e    1.030659
dtype: float64
```

This is most definitely a “practicality beats purity” sort of thing, but it is something to watch out for if you expect label-based slicing to behave exactly in the way that standard Python integer slicing works.

### 13.6.4 Indexing potentially changes underlying Series dtype

The different indexing operation can potentially change the dtype of a Series.

```
In [198]: series1 = pd.Series([1, 2, 3])

In [199]: series1.dtype
Out[199]: dtype('int64')

In [200]: res = series1.reindex([0, 4])
```

(continues on next page)



(continued from previous page)

```
In [201]: res.dtype
Out[201]: dtype('float64')

In [202]: res
\\Out[202]:
0      1.0
4      NaN
dtype: float64
```

```
In [203]: series2 = pd.Series([True])

In [204]: series2.dtype
Out[204]: dtype('bool')

In [205]: res = series2.reindex_like(series1)

In [206]: res.dtype
Out[206]: dtype('O')

In [207]: res
\\Out[207]:
0      True
1      NaN
2      NaN
dtype: object
```

This is because the (re)indexing operations above silently inserts NaNs and the dtype changes accordingly. This can cause some issues when using numpy ufuncs such as `numpy.logical_and`.

See the [this old issue](#) for a more detailed discussion.



## COMPUTATIONAL TOOLS

### 14.1 Statistical Functions

#### 14.1.1 Percent Change

`Series`, `DataFrame`, and `Panel` all have a method `pct_change()` to compute the percent change over a given number of periods (using `fill_method` to fill NA/null values *before* computing the percent change).

```
In [1]: ser = pd.Series(np.random.randn(8))
```

```
In [2]: ser.pct_change()
```

```
Out [2]:
```

```
0      NaN
1   -1.602976
2    4.334938
3   -0.247456
4   -2.067345
5   -1.142903
6   -1.688214
7   -9.759729
dtype: float64
```

```
In [3]: df = pd.DataFrame(np.random.randn(10, 4))
```

```
In [4]: df.pct_change(periods=3)
```

```
Out [4]:
```

```
      0      1      2      3
0     NaN     NaN     NaN     NaN
1     NaN     NaN     NaN     NaN
2     NaN     NaN     NaN     NaN
3 -0.218320 -1.054001  1.987147 -0.510183
4 -0.439121 -1.816454  0.649715 -4.822809
5 -0.127833 -3.042065 -5.866604 -1.776977
6 -2.596833 -1.959538 -2.111697 -3.798900
7 -0.117826 -2.169058  0.036094 -0.067696
8  2.492606 -1.357320 -1.205802 -1.558697
9 -1.012977  2.324558 -1.003744 -0.371806
```

#### 14.1.2 Covariance

`Series.cov()` can be used to compute covariance between series (excluding missing values).

```
In [5]: s1 = pd.Series(np.random.randn(1000))

In [6]: s2 = pd.Series(np.random.randn(1000))

In [7]: s1.cov(s2)
Out[7]: 0.00068010881743110871
```

Analogously, `DataFrame.cov()` to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

**Note:** Assuming the missing data are missing at random this results in an estimate for the covariance matrix which is unbiased. However, for many applications this estimate may not be acceptable because the estimated covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimated correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See [Estimation of covariance matrices](#) for more details.

```
In [8]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e']
↳)

In [9]: frame.cov()
Out[9]:
```

|   | a         | b         | c         | d         | e         |
|---|-----------|-----------|-----------|-----------|-----------|
| a | 1.000882  | -0.003177 | -0.002698 | -0.006889 | 0.031912  |
| b | -0.003177 | 1.024721  | 0.000191  | 0.009212  | 0.000857  |
| c | -0.002698 | 0.000191  | 0.950735  | -0.031743 | -0.005087 |
| d | -0.006889 | 0.009212  | -0.031743 | 1.002983  | -0.047952 |
| e | 0.031912  | 0.000857  | -0.005087 | -0.047952 | 1.042487  |

`DataFrame.cov` also supports an optional `min_periods` keyword that specifies the required minimum number of observations for each column pair in order to have a valid result.

```
In [10]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])

In [11]: frame.loc[frame.index[:5], 'a'] = np.nan

In [12]: frame.loc[frame.index[5:10], 'b'] = np.nan

In [13]: frame.cov()
Out[13]:
```

|   | a         | b         | c        |
|---|-----------|-----------|----------|
| a | 1.123670  | -0.412851 | 0.018169 |
| b | -0.412851 | 1.154141  | 0.305260 |
| c | 0.018169  | 0.305260  | 1.301149 |

```
In [14]: frame.cov(min_periods=12)
↳
```

|   | a        | b        | c        |
|---|----------|----------|----------|
| a | 1.123670 | NaN      | 0.018169 |
| b | NaN      | 1.154141 | 0.305260 |
| c | 0.018169 | 0.305260 | 1.301149 |

### 14.1.3 Correlation

Correlation may be computed using the `corr()` method. Using the `method` parameter, several methods for computing correlations are provided:

| Method name          | Description                           |
|----------------------|---------------------------------------|
| pearson<br>(default) | Standard correlation coefficient      |
| kendall              | Kendall Tau correlation coefficient   |
| spearman             | Spearman rank correlation coefficient |

All of these are currently computed using pairwise complete observations. Wikipedia has articles covering the above correlation coefficients:

- [Pearson correlation coefficient](#)
- [Kendall rank correlation coefficient](#)
- [Spearman's rank correlation coefficient](#)

**Note:** Please see the *caveats* associated with this method of calculating correlation matrices in the *covariance section*.

```
In [15]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=['a', 'b', 'c', 'd',
↳ 'e'])

In [16]: frame.iloc[:,2] = np.nan

# Series with Series
In [17]: frame['a'].corr(frame['b'])
Out[17]: 0.013479040400098794

In [18]: frame['a'].corr(frame['b'], method='spearman')
Out[18]: -0.0072898851595406371

# Pairwise correlation of DataFrame columns
In [19]: frame.corr()
Out[19]:
```

|   | a         | b         | c         | d         | e         |
|---|-----------|-----------|-----------|-----------|-----------|
| a | 1.000000  | 0.013479  | -0.049269 | -0.042239 | -0.028525 |
| b | 0.013479  | 1.000000  | -0.020433 | -0.011139 | 0.005654  |
| c | -0.049269 | -0.020433 | 1.000000  | 0.018587  | -0.054269 |
| d | -0.042239 | -0.011139 | 0.018587  | 1.000000  | -0.017060 |
| e | -0.028525 | 0.005654  | -0.054269 | -0.017060 | 1.000000  |

Note that non-numeric columns will be automatically excluded from the correlation calculation.

Like `cov`, `corr` also supports the optional `min_periods` keyword:

```
In [20]: frame = pd.DataFrame(np.random.randn(20, 3), columns=['a', 'b', 'c'])

In [21]: frame.loc[frame.index[:5], 'a'] = np.nan

In [22]: frame.loc[frame.index[5:10], 'b'] = np.nan

In [23]: frame.corr()
Out[23]:
```

(continues on next page)

(continued from previous page)

```

      a      b      c
a  1.000000 -0.121111  0.069544
b -0.121111  1.000000  0.051742
c  0.069544  0.051742  1.000000

```

```
In [24]: frame.corr(min_periods=12)
```

```

////////////////////////////////////Out
↪
      a      b      c
a  1.000000    NaN  0.069544
b      NaN  1.000000  0.051742
c  0.069544  0.051742  1.000000

```

A related method `corrwith()` is implemented on `DataFrame` to compute the correlation between like-labeled Series contained in different `DataFrame` objects.

```
In [25]: index = ['a', 'b', 'c', 'd', 'e']
```

```
In [26]: columns = ['one', 'two', 'three', 'four']
```

```
In [27]: df1 = pd.DataFrame(np.random.randn(5, 4), index=index, columns=columns)
```

```
In [28]: df2 = pd.DataFrame(np.random.randn(4, 4), index=index[:4], columns=columns)
```

```
In [29]: df1.corrwith(df2)
```

```
Out[29]:
```

```

one      -0.125501
two       -0.493244
three      0.344056
four       0.004183
dtype: float64

```

```
In [30]: df2.corrwith(df1, axis=1)
```

```

////////////////////////////////////Out
↪
a      -0.675817
b       0.458296
c       0.190809
d      -0.186275
e           NaN
dtype: float64

```

### 14.1.4 Data ranking

The `rank()` method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```
In [31]: s = pd.Series(np.random.randn(5), index=list('abcde'))
```

```
In [32]: s['d'] = s['b'] # so there's a tie
```

```
In [33]: s.rank()
```

```

Out[33]:
a      5.0
b      2.5
c      1.0

```

(continues on next page)

(continued from previous page)

```
d    2.5
e    4.0
dtype: float64
```

`rank()` is also a DataFrame method and can rank either the rows (`axis=0`) or the columns (`axis=1`). NaN values are excluded from the ranking.

```
In [34]: df = pd.DataFrame(np.random.randn(10, 6))
```

```
In [35]: df[4] = df[2][:5] # some ties
```

```
In [36]: df
```

```
Out[36]:
```

```
      0         1         2         3         4         5
0 -0.904948 -1.163537 -1.457187  0.135463 -1.457187  0.294650
1 -0.976288 -0.244652 -0.748406 -0.999601 -0.748406 -0.800809
2  0.401965  1.460840  1.256057  1.308127  1.256057  0.876004
3  0.205954  0.369552 -0.669304  0.038378 -0.669304  1.140296
4 -0.477586 -0.730705 -1.129149 -0.601463 -1.129149 -0.211196
5 -1.092970 -0.689246  0.908114  0.204848         NaN  0.463347
6  0.376892  0.959292  0.095572 -0.593740         NaN -0.069180
7 -1.002601  1.957794 -0.120708  0.094214         NaN -1.467422
8 -0.547231  0.664402 -0.519424 -0.073254         NaN -1.263544
9 -0.250277 -0.237428 -1.056443  0.419477         NaN  1.375064
```

```
In [37]: df.rank(1)
```

```

////////////////////////////////////
↪
      0         1         2         3         4         5
0  4.0  3.0  1.5  5.0  1.5  6.0
1  2.0  6.0  4.5  1.0  4.5  3.0
2  1.0  6.0  3.5  5.0  3.5  2.0
3  4.0  5.0  1.5  3.0  1.5  6.0
4  5.0  3.0  1.5  4.0  1.5  6.0
5  1.0  2.0  5.0  3.0  NaN  4.0
6  4.0  5.0  3.0  1.0  NaN  2.0
7  2.0  5.0  3.0  4.0  NaN  1.0
8  2.0  5.0  3.0  4.0  NaN  1.0
9  2.0  3.0  1.0  4.0  NaN  5.0
```

`rank` optionally takes a parameter `ascending` which by default is `true`; when `false`, data is reverse-ranked, with larger values assigned a smaller rank.

`rank` supports different tie-breaking methods, specified with the `method` parameter:

- `average` : average rank of tied group
- `min` : lowest rank in the group
- `max` : highest rank in the group
- `first` : ranks assigned in the order they appear in the array

## 14.2 Window Functions

For working with data, a number of window functions are provided for computing common *window* or *rolling* statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and

kurtosis.

The `rolling()` and `expanding()` functions can be used directly from `DataFrameGroupBy` objects, see the [groupby docs](#).

**Note:** The API for window statistics is quite similar to the way one works with `GroupBy` objects, see the documentation [here](#).

We work with rolling, expanding and exponentially weighted data through the corresponding objects, Rolling, Expanding and EWM.

```
In [38]: s = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
↳ periods=1000))

In [39]: s = s.cumsum()

In [40]: s
Out[40]:
2000-01-01    -0.268824
2000-01-02    -1.771855
2000-01-03    -0.818003
2000-01-04    -0.659244
2000-01-05    -1.942133
2000-01-06    -1.869391
2000-01-07     0.563674
...
2002-09-20   -68.233054
2002-09-21   -66.765687
2002-09-22   -67.457323
2002-09-23   -69.253182
2002-09-24   -70.296818
2002-09-25   -70.844674
2002-09-26   -72.475016
Freq: D, Length: 1000, dtype: float64
```

These are created from methods on `Series` and `DataFrame`.

```
In [41]: r = s.rolling(window=60)

In [42]: r
Out[42]: Rolling [window=60,center=False,axis=0]
```

These objects provide tab-completion of the available methods and properties.

```
In [14]: r.
r.agg          r.apply          r.count          r.exclusions  r.max          r.median        r.
↳ name        r.skew          r.sum            r.kurt        r.mean         r.min          r.
r.agg         r.corr          r.cov            r.kurt        r.mean         r.min          r.
↳ quantile    r.std          r.var
```

Generally these methods all have the same interface. They all accept the following arguments:

- `window`: size of moving window
- `min_periods`: threshold of non-null data points to require (otherwise result is NA)
- `center`: boolean, whether to set the labels at the center (default is `False`)

We can then call methods on these rolling objects. These return like-indexed objects:



```
In [43]: r.mean()
```

```
Out [43]:
```

```
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05      NaN
2000-01-06      NaN
2000-01-07      NaN
```

```
...
```

```
2002-09-20    -62.694135
2002-09-21    -62.812190
2002-09-22    -62.914971
2002-09-23    -63.061867
2002-09-24    -63.213876
2002-09-25    -63.375074
2002-09-26    -63.539734
```

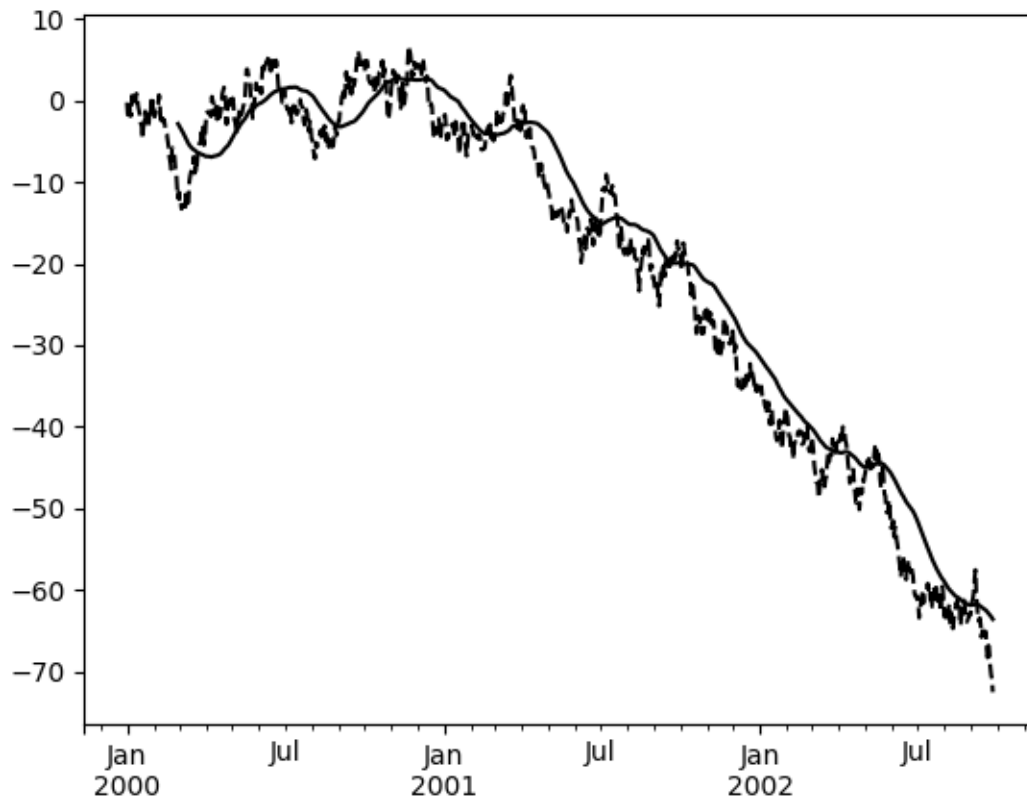
```
Freq: D, Length: 1000, dtype: float64
```

```
In [44]: s.plot(style='k--')
```

```
Out [44]: <matplotlib.axes._subplots.AxesSubplot at 0x1a24661e80>
```

```
In [45]: r.mean().plot(style='k')
```

```
Out [45]: <matplotlib.axes._subplots.AxesSubplot at 0x1a24661e80>
```

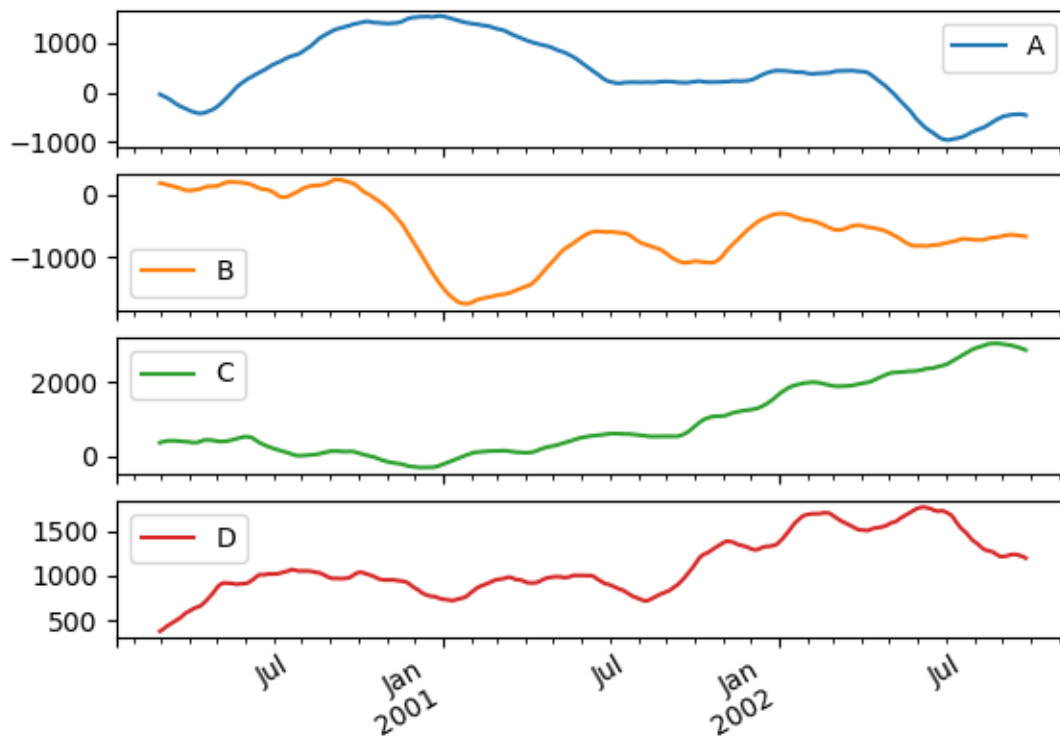


They can also be applied to DataFrame objects. This is really just syntactic sugar for applying the moving window operator to all of the DataFrame's columns:

```
In [46]: df = pd.DataFrame(np.random.randn(1000, 4),
.....:                    index=pd.date_range('1/1/2000', periods=1000),
.....:                    columns=['A', 'B', 'C', 'D'])
.....:

In [47]: df = df.cumsum()

In [48]: df.rolling(window=60).sum().plot(subplots=True)
Out[48]:
array([<matplotlib.axes._subplots.AxesSubplot object at 0x1a24757e48>,
      <matplotlib.axes._subplots.AxesSubplot object at 0x1a24780c88>,
      <matplotlib.axes._subplots.AxesSubplot object at 0x1a247aae48>,
      <matplotlib.axes._subplots.AxesSubplot object at 0x1a247db048>], dtype=object)
```



### 14.2.1 Method Summary

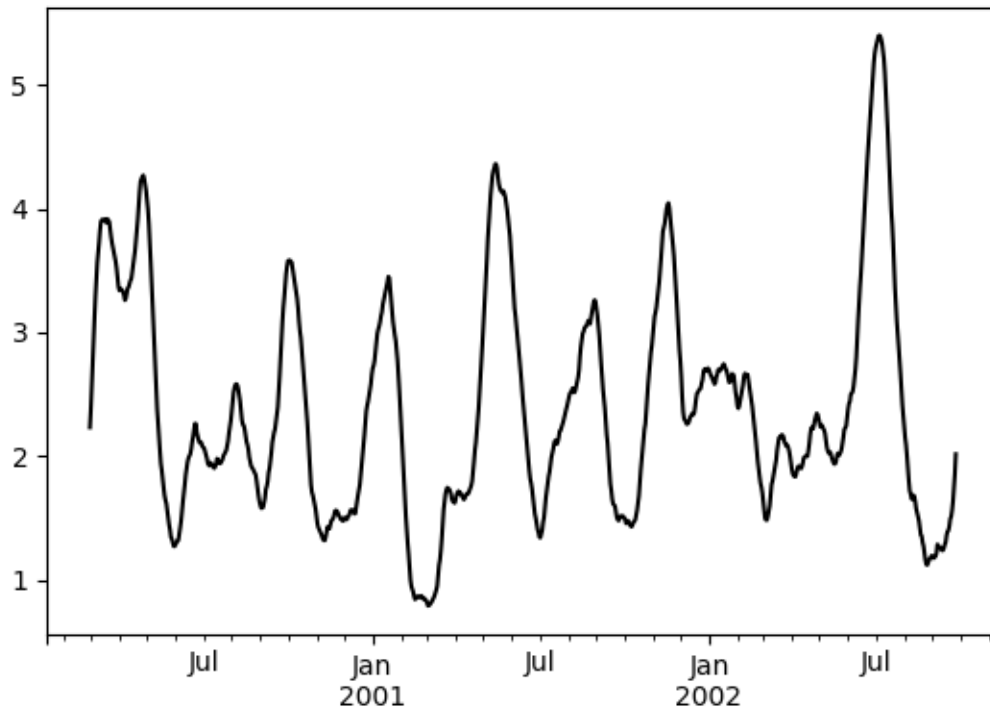
We provide a number of common statistical functions:

| Method                  | Description                                |
|-------------------------|--|
| <code>count()</code>    | Number of non-null observations            |
| <code>sum()</code>      | Sum of values                              |
| <code>mean()</code>     | Mean of values                             |
| <code>median()</code>   | Arithmetic median of values                |
| <code>min()</code>      | Minimum                                    |
| <code>max()</code>      | Maximum                                    |
| <code>std()</code>      | Bessel-corrected sample standard deviation |
| <code>var()</code>      | Unbiased variance                          |
| <code>skew()</code>     | Sample skewness (3rd moment)               |
| <code>kurt()</code>     | Sample kurtosis (4th moment)               |
| <code>quantile()</code> | Sample quantile (value at %)               |
| <code>apply()</code>    | Generic apply                              |
| <code>cov()</code>      | Unbiased covariance (binary)               |
| <code>corr()</code>     | Correlation (binary)                       |

The `apply()` function takes an extra `func` argument and performs generic rolling computations. The `func` argument should be a single function that produces a single value from an ndarray input. Suppose we wanted to compute the mean absolute deviation on a rolling basis:

```
In [49]: mad = lambda x: np.fabs(x - x.mean()).mean()

In [50]: s.rolling(window=60).apply(mad, raw=True).plot(style='k')
Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x1a24b21630>
```



### 14.2.2 Rolling Windows

Passing `win_type` to `.rolling` generates a generic rolling window computation, that is weighted according to the `win_type`. The following methods are available:

| Method              | Description    |
|---------------------|----------------|
| <code>sum()</code>  | Sum of values  |
| <code>mean()</code> | Mean of values |

The weights used in the window are specified by the `win_type` keyword. The list of recognized types are the [scipy.signal window functions](#):

- `boxcar`
- `triang`
- `blackman`
- `hamming`
- `bartlett`
- `parzen`
- `bohman`

- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general\_gaussian (needs power, width)
- slepian (needs width).

```
In [51]: ser = pd.Series(np.random.randn(10), index=pd.date_range('1/1/2000',
↳ periods=10))
```

```
In [52]: ser.rolling(window=5, win_type='triang').mean()
```

```
Out[52]:
```

```
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05   -1.037870
2000-01-06   -0.767705
2000-01-07   -0.383197
2000-01-08   -0.395513
2000-01-09   -0.558440
2000-01-10   -0.672416
Freq: D, dtype: float64
```

Note that the boxcar window is equivalent to `mean()`.

```
In [53]: ser.rolling(window=5, win_type='boxcar').mean()
```

```
Out[53]:
```

```
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

```
In [54]: ser.rolling(window=5).mean()
```

```
////////////////////////////////////
```

```
↳
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05   -0.841164
2000-01-06   -0.779948
2000-01-07   -0.565487
2000-01-08   -0.502815
2000-01-09   -0.553755
2000-01-10   -0.472211
Freq: D, dtype: float64
```

For some windowing functions, additional parameters must be specified:

```
In [55]: ser.rolling(window=5, win_type='gaussian').mean(std=0.1)
Out [55]:
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05    -1.309989
2000-01-06    -1.153000
2000-01-07     0.606382
2000-01-08    -0.681101
2000-01-09    -0.289724
2000-01-10    -0.996632
Freq: D, dtype: float64
```

---

**Note:** For `.sum()` with a `win_type`, there is no normalization done to the weights for the window. Passing custom weights of `[1, 1, 1]` will yield a different result than passing weights of `[2, 2, 2]`, for example. When passing a `win_type` instead of explicitly specifying the weights, the weights are already normalized so that the largest weight is 1.

In contrast, the nature of the `.mean()` calculation is such that the weights are normalized with respect to each other. Weights of `[1, 1, 1]` and `[2, 2, 2]` yield the same result.

---

### 14.2.3 Time-aware Rolling

New in version 0.19.0.

New in version 0.19.0 are the ability to pass an offset (or convertible) to a `.rolling()` method and have it produce variable sized windows based on the passed time window. For each time point, this includes all preceding values occurring within the indicated time delta.

This can be particularly useful for a non-regular time frequency index.

```
In [56]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
.....:                      index=pd.date_range('20130101 09:00:00', periods=5, freq=
↳ 's'))
.....:

In [57]: dft
Out [57]:
                B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:01  1.0
2013-01-01 09:00:02  2.0
2013-01-01 09:00:03  NaN
2013-01-01 09:00:04  4.0
```

This is a regular frequency index. Using an integer window parameter works to roll along the window frequency.

```
In [58]: dft.rolling(2).sum()
Out [58]:
                B
2013-01-01 09:00:00  NaN
2013-01-01 09:00:01  1.0
```

(continues on next page)

(continued from previous page)

```

2013-01-01 09:00:02    3.0
2013-01-01 09:00:03    NaN
2013-01-01 09:00:04    NaN

```

```
In [59]: dft.rolling(2, min_periods=1).sum()
```

```

////////////////////////////////////
↪
      B
2013-01-01 09:00:00    0.0
2013-01-01 09:00:01    1.0
2013-01-01 09:00:02    3.0
2013-01-01 09:00:03    2.0
2013-01-01 09:00:04    4.0

```

Specifying an offset allows a more intuitive specification of the rolling frequency.

```
In [60]: dft.rolling('2s').sum()
```

```
Out [60]:
```

```

      B
2013-01-01 09:00:00    0.0
2013-01-01 09:00:01    1.0
2013-01-01 09:00:02    3.0
2013-01-01 09:00:03    2.0
2013-01-01 09:00:04    4.0

```

Using a non-regular, but still monotonic index, rolling with an integer window does not impart any special calculation.

```

In [61]: dft = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
.....:                      index = pd.Index([pd.Timestamp('20130101 09:00:00'),
.....:                                         pd.Timestamp('20130101 09:00:02'),
.....:                                         pd.Timestamp('20130101 09:00:03'),
.....:                                         pd.Timestamp('20130101 09:00:05'),
.....:                                         pd.Timestamp('20130101 09:00:06')],
.....:                                         name='foo'))

```

```
In [62]: dft
```

```
Out [62]:
```

```

      B
foo
2013-01-01 09:00:00    0.0
2013-01-01 09:00:02    1.0
2013-01-01 09:00:03    2.0
2013-01-01 09:00:05    NaN
2013-01-01 09:00:06    4.0

```

```
In [63]: dft.rolling(2).sum()
```

```

////////////////////////////////////
↪
      B
foo
2013-01-01 09:00:00    NaN
2013-01-01 09:00:02    1.0
2013-01-01 09:00:03    3.0
2013-01-01 09:00:05    NaN
2013-01-01 09:00:06    NaN

```

Using the time-specification generates variable windows for this sparse data.

```
In [64]: dft.rolling('2s').sum()
Out[64]:
```

|            |          | B   |
|------------|----------|-----|
| foo        |          |     |
| 2013-01-01 | 09:00:00 | 0.0 |
| 2013-01-01 | 09:00:02 | 1.0 |
| 2013-01-01 | 09:00:03 | 3.0 |
| 2013-01-01 | 09:00:05 | NaN |
| 2013-01-01 | 09:00:06 | 4.0 |

Furthermore, we now allow an optional `on` parameter to specify a column (rather than the default of the index) in a `DataFrame`.

```
In [65]: dft = dft.reset_index()
```

```
In [66]: dft
```

```
Out [66]:
```

|   |                     | foo | B |
|---|---------------------|-----|---|
| 0 | 2013-01-01 09:00:00 | 0.0 |   |
| 1 | 2013-01-01 09:00:02 | 1.0 |   |
| 2 | 2013-01-01 09:00:03 | 2.0 |   |
| 3 | 2013-01-01 09:00:05 | NaN |   |
| 4 | 2013-01-01 09:00:06 | 4.0 |   |

```
In [67]: dft.rolling('2s', on='foo').sum()
```

|   |            |          | foo | B |
|---|------------|----------|-----|---|
| 0 | 2013-01-01 | 09:00:00 | 0.0 |   |
| 1 | 2013-01-01 | 09:00:02 | 1.0 |   |
| 2 | 2013-01-01 | 09:00:03 | 3.0 |   |
| 3 | 2013-01-01 | 09:00:05 | NaN |   |
| 4 | 2013-01-01 | 09:00:06 | 4.0 |   |

#### 14.2.4 Rolling Window Endpoints

New in version 0.20.0.

The inclusion of the interval endpoints in rolling window calculations can be specified with the `closed` parameter:

| closed  | Description          | Default for        |
|---------|----------------------|--------------------|
| right   | close right endpoint | time-based windows |
| left    | close left endpoint  |                    |
| both    | close both endpoints | fixed windows      |
| neither | open endpoints       |                    |

For example, having the right endpoint open is useful in many problems that require that there is no contamination from present information back to past information. This allows the rolling window to compute statistics “up to that point in time”, but not including that point in time.

[illegible]

(continues on next page)



(continued from previous page)

```

.....:         pd.Timestamp('20130101 09:00:03'),
.....:         pd.Timestamp('20130101 09:00:04'),
.....:         pd.Timestamp('20130101 09:00:06')]})
.....:

In [69]: df["right"] = df.rolling('2s', closed='right').x.sum() # default

In [70]: df["both"] = df.rolling('2s', closed='both').x.sum()

In [71]: df["left"] = df.rolling('2s', closed='left').x.sum()

In [72]: df["neither"] = df.rolling('2s', closed='neither').x.sum()

In [73]: df
Out[73]:

```

|                     | x | right | both | left | neither |
|---------------------|---|-------|------|------|---------|
| 2013-01-01 09:00:01 | 1 | 1.0   | 1.0  | NaN  | NaN     |
| 2013-01-01 09:00:02 | 1 | 2.0   | 2.0  | 1.0  | 1.0     |
| 2013-01-01 09:00:03 | 1 | 2.0   | 3.0  | 2.0  | 1.0     |
| 2013-01-01 09:00:04 | 1 | 2.0   | 3.0  | 2.0  | 1.0     |
| 2013-01-01 09:00:06 | 1 | 1.0   | 2.0  | 1.0  | NaN     |

Currently, this feature is only implemented for time-based windows. For fixed windows, the `closed` parameter cannot be set and the rolling window will always have both endpoints closed.

### 14.2.5 Time-aware Rolling vs. Resampling

Using `.rolling()` with a time-based index is quite similar to *resampling*. They both operate and perform reductive operations on time-indexed pandas objects.

When using `.rolling()` with an offset. The offset is a time-delta. Take a backwards-in-time looking window, and aggregate all of the values in that window (including the end-point, but not the start-point). This is the new value at that point in the result. These are variable sized windows in time-space for each point of the input. You will get a same sized result as the input.

When using `.resample()` with an offset. Construct a new index that is the frequency of the offset. For each frequency bin, aggregate points from the input within a backwards-in-time looking window that fall in that bin. The result of this aggregation is the output for that frequency point. The windows are fixed size in the frequency space. Your result will have the shape of a regular frequency between the min and the max of the original input object.

To summarize, `.rolling()` is a time-based window operation, while `.resample()` is a frequency-based window operation.

### 14.2.6 Centering Windows

By default the labels are set to the right edge of the window, but a `center` keyword is available so the labels can be set at the center.

```

In [74]: ser.rolling(window=5).mean()
Out[74]:
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN

```

(continues on next page)

(continued from previous page)

```

2000-01-05    -0.841164
2000-01-06    -0.779948
2000-01-07    -0.565487
2000-01-08    -0.502815
2000-01-09    -0.553755
2000-01-10    -0.472211
Freq: D, dtype: float64

```

```
In [75]: ser.rolling(window=5, center=True).mean()
```

```

↪
2000-01-01      NaN
2000-01-02      NaN
2000-01-03    -0.841164
2000-01-04    -0.779948
2000-01-05    -0.565487
2000-01-06    -0.502815
2000-01-07    -0.553755
2000-01-08    -0.472211
2000-01-09      NaN
2000-01-10      NaN
Freq: D, dtype: float64

```

## 14.2.7 Binary Window Functions

`cov()` and `corr()` can compute moving window statistics about two Series or any combination of DataFrame/Series or DataFrame/DataFrame. Here is the behavior in each case:

- two Series: compute the statistic for the pairing.
- DataFrame/Series: compute the statistics for each column of the DataFrame with the passed Series, thus returning a DataFrame.
- DataFrame/DataFrame: by default compute the statistic for matching column names, returning a DataFrame. If the keyword argument `pairwise=True` is passed then computes the statistic for each pair of columns, returning a MultiIndexed DataFrame whose index are the dates in question (see [the next section](#)).

For example:

```

In [76]: df = pd.DataFrame(np.random.randn(1000, 4),
.....:                     index=pd.date_range('1/1/2000', periods=1000),
.....:                     columns=['A', 'B', 'C', 'D'])
.....:

In [77]: df = df.cumsum()

In [78]: df2 = df[:20]

In [79]: df2.rolling(window=5).corr(df2['B'])
Out [79]:
           A      B      C      D
2000-01-01  NaN  NaN  NaN  NaN
2000-01-02  NaN  NaN  NaN  NaN
2000-01-03  NaN  NaN  NaN  NaN
2000-01-04  NaN  NaN  NaN  NaN

```

(continues on next page)

(continued from previous page)

```

2000-01-05  0.768775  1.0 -0.977990  0.800252
2000-01-06  0.744106  1.0 -0.967912  0.830021
2000-01-07  0.683257  1.0 -0.928969  0.384916
...
2000-01-14 -0.392318  1.0  0.570240 -0.591056
2000-01-15  0.017217  1.0  0.649900 -0.896258
2000-01-16  0.691078  1.0  0.807450 -0.939302
2000-01-17  0.274506  1.0  0.582601 -0.902954
2000-01-18  0.330459  1.0  0.515707 -0.545268
2000-01-19  0.046756  1.0 -0.104334 -0.419799
2000-01-20 -0.328241  1.0 -0.650974 -0.777777

[20 rows x 4 columns]

```

## 14.2.8 Computing rolling pairwise covariances and correlations

**Warning:** Prior to version 0.20.0 if `pairwise=True` was passed, a `Panel` would be returned. This will now return a 2-level `MultiIndexed DataFrame`, see the [whatsnew here](#).

In financial data analysis and other fields it's common to compute covariance and correlation matrices for a collection of time series. Often one is also interested in moving-window covariance and correlation matrices. This can be done by passing the `pairwise` keyword argument, which in the case of `DataFrame` inputs will yield a `MultiIndexed DataFrame` whose index are the dates in question. In the case of a single `DataFrame` argument the `pairwise` argument can even be omitted:

**Note:** Missing values are ignored and each entry is computed using the pairwise complete observations. Please see the [covariance section](#) for *caveats* associated with this method of calculating covariance and correlation matrices.

```

In [80]: covs = df[['B', 'C', 'D']].rolling(window=50).cov(df[['A', 'B', 'C']],
↳ pairwise=True)

```

```

In [81]: covs.loc['2002-09-22':]

```

```

Out[81]:

```

```

          B          C          D
2002-09-22 A  1.367467  8.676734 -8.047366
          B  3.067315  0.865946 -1.052533
          C  0.865946  7.739761 -4.943924
2002-09-23 A  0.910343  8.669065 -8.443062
          B  2.625456  0.565152 -0.907654
          C  0.565152  7.825521 -5.367526
2002-09-24 A  0.463332  8.514509 -8.776514
          B  2.306695  0.267746 -0.732186
          C  0.267746  7.771425 -5.696962
2002-09-25 A  0.467976  8.198236 -9.162599
          B  2.307129  0.267287 -0.754080
          C  0.267287  7.466559 -5.822650
2002-09-26 A  0.545781  7.899084 -9.326238
          B  2.311058  0.322295 -0.844451
          C  0.322295  7.038237 -5.684445

```

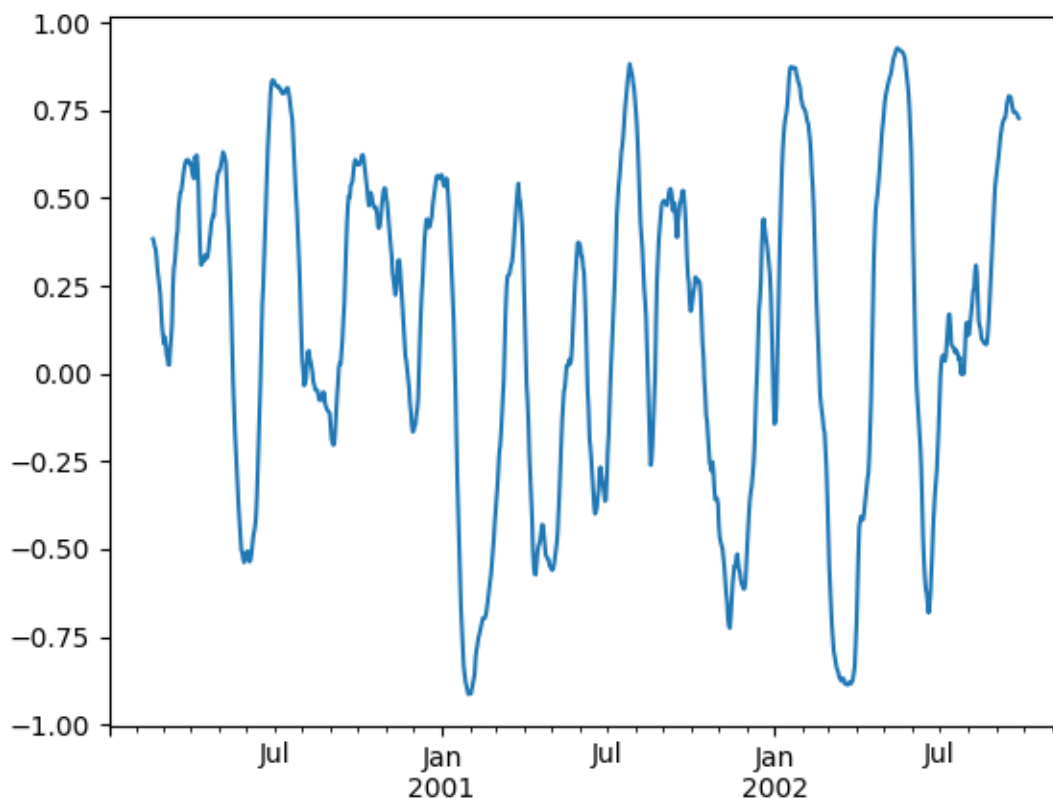
```
In [82]: correls = df.rolling(window=50).corr()
In [83]: correls.loc['2002-09-22':]
Out[83]:
```

|            |     | A         | B         | C         | D         |
|------------|-----|-----------|-----------|-----------|-----------|
| 2002-09-22 | A   | 1.000000  | 0.186397  | 0.744551  | -0.769767 |
|            | B   | 0.186397  | 1.000000  | 0.177725  | -0.240802 |
|            | C   | 0.744551  | 0.177725  | 1.000000  | -0.712051 |
|            | D   | -0.769767 | -0.240802 | -0.712051 | 1.000000  |
| 2002-09-23 | A   | 1.000000  | 0.134723  | 0.743113  | -0.758758 |
|            | B   | 0.134723  | 1.000000  | 0.124683  | -0.209934 |
|            | C   | 0.743113  | 0.124683  | 1.000000  | -0.719088 |
| ...        | ... | ...       | ...       | ...       | ...       |
| 2002-09-25 | B   | 0.075157  | 1.000000  | 0.064399  | -0.164179 |
|            | C   | 0.731888  | 0.064399  | 1.000000  | -0.704686 |
|            | D   | -0.739160 | -0.164179 | -0.704686 | 1.000000  |
| 2002-09-26 | A   | 1.000000  | 0.087756  | 0.727792  | -0.736562 |
|            | B   | 0.087756  | 1.000000  | 0.079913  | -0.179477 |
|            | C   | 0.727792  | 0.079913  | 1.000000  | -0.692303 |
|            | D   | -0.736562 | -0.179477 | -0.692303 | 1.000000  |

```
[20 rows x 4 columns]
```

You can efficiently retrieve the time series of correlations between two columns by reshaping and indexing:

```
In [84]: correls.unstack(1)[('A', 'C')].plot()
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x1c25247630>
```



## 14.3 Aggregation

Once the `Rolling`, `Expanding` or `EWM` objects have been created, several methods are available to perform multiple computations on the data. These operations are similar to the *aggregating API*, *groupby API*, and *resample API*.

```
In [85]: dfa = pd.DataFrame(np.random.randn(1000, 3),
.....:                     index=pd.date_range('1/1/2000', periods=1000),
.....:                     columns=['A', 'B', 'C'])
.....:

In [86]: r = dfa.rolling(window=60, min_periods=1)

In [87]: r
Out[87]: Rolling [window=60, min_periods=1, center=False, axis=0]
```

We can aggregate by passing a function to the entire `DataFrame`, or select a `Series` (or multiple `Series`) via standard `__getitem__`.

```
In [88]: r.aggregate(np.sum)
Out[88]:
```

|            | A         | B         | C         |
|------------|-----------|-----------|-----------|
| 2000-01-01 | -0.289838 | -0.370545 | -1.284206 |
| 2000-01-02 | -0.216612 | -1.675528 | -1.169415 |

(continues on next page)

(continued from previous page)

```

2000-01-03    1.154661   -1.634017   -1.566620
2000-01-04    2.969393   -4.003274   -1.816179
2000-01-05    4.690630   -4.682017   -2.717209
2000-01-06    3.880630   -4.447700   -1.078947
2000-01-07    4.001957   -2.884072   -3.116903
...
2002-09-20    2.652493  -10.528875    9.867805
2002-09-21    0.844497   -9.280944    9.522649
2002-09-22    2.860036   -9.270337    6.415245
2002-09-23    3.510163   -8.151439    5.177219
2002-09-24    6.524983  -10.168078    5.792639
2002-09-25    6.409626   -9.956226    5.704050
2002-09-26    5.093787   -7.074515    6.905823

```

```
[1000 rows x 3 columns]
```

```
In [89]: r['A'].aggregate(np.sum)
```

```

////////////////////////////////////
↪
2000-01-01    -0.289838
2000-01-02    -0.216612
2000-01-03     1.154661
2000-01-04     2.969393
2000-01-05     4.690630
2000-01-06     3.880630
2000-01-07     4.001957
...
2002-09-20     2.652493
2002-09-21     0.844497
2002-09-22     2.860036
2002-09-23     3.510163
2002-09-24     6.524983
2002-09-25     6.409626
2002-09-26     5.093787
Freq: D, Name: A, Length: 1000, dtype: float64

```

```
In [90]: r[['A', 'B']].aggregate(np.sum)
```

```

////////////////////////////////////
↪
           A           B
2000-01-01  -0.289838  -0.370545
2000-01-02  -0.216612  -1.675528
2000-01-03   1.154661  -1.634017
2000-01-04   2.969393  -4.003274
2000-01-05   4.690630  -4.682017
2000-01-06   3.880630  -4.447700
2000-01-07   4.001957  -2.884072
...
2002-09-20   2.652493 -10.528875
2002-09-21   0.844497  -9.280944
2002-09-22   2.860036  -9.270337
2002-09-23   3.510163  -8.151439
2002-09-24   6.524983 -10.168078
2002-09-25   6.409626  -9.956226
2002-09-26   5.093787  -7.074515

```

```
[1000 rows x 2 columns]
```

As you can see, the result of the aggregation will have the selected columns, or all columns if none are selected.

### 14.3.1 Applying multiple functions

With windowed Series you can also pass a list of functions to do aggregation with, outputting a DataFrame:

```
In [91]: r['A'].agg([np.sum, np.mean, np.std])
```

```
Out [91]:
```

|            | sum       | mean      | std      |
|------------|-----------|-----------|----------|
| 2000-01-01 | -0.289838 | -0.289838 | NaN      |
| 2000-01-02 | -0.216612 | -0.108306 | 0.256725 |
| 2000-01-03 | 1.154661  | 0.384887  | 0.873311 |
| 2000-01-04 | 2.969393  | 0.742348  | 1.009734 |
| 2000-01-05 | 4.690630  | 0.938126  | 0.977914 |
| 2000-01-06 | 3.880630  | 0.646772  | 1.128883 |
| 2000-01-07 | 4.001957  | 0.571708  | 1.049487 |
| ...        | ...       | ...       | ...      |
| 2002-09-20 | 2.652493  | 0.044208  | 1.164919 |
| 2002-09-21 | 0.844497  | 0.014075  | 1.148231 |
| 2002-09-22 | 2.860036  | 0.047667  | 1.132051 |
| 2002-09-23 | 3.510163  | 0.058503  | 1.134296 |
| 2002-09-24 | 6.524983  | 0.108750  | 1.144204 |
| 2002-09-25 | 6.409626  | 0.106827  | 1.142913 |
| 2002-09-26 | 5.093787  | 0.084896  | 1.151416 |

```
[1000 rows x 3 columns]
```

On a windowed DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [92]: r.agg([np.sum, np.mean])
```

```
Out [92]:
```

|            | A         |           | B          |           | C         |           |
|------------|-----------|-----------|------------|-----------|-----------|-----------|
|            | sum       | mean      | sum        | mean      | sum       | mean      |
| 2000-01-01 | -0.289838 | -0.289838 | -0.370545  | -0.370545 | -1.284206 | -1.284206 |
| 2000-01-02 | -0.216612 | -0.108306 | -1.675528  | -0.837764 | -1.169415 | -0.584708 |
| 2000-01-03 | 1.154661  | 0.384887  | -1.634017  | -0.544672 | -1.566620 | -0.522207 |
| 2000-01-04 | 2.969393  | 0.742348  | -4.003274  | -1.000819 | -1.816179 | -0.454045 |
| 2000-01-05 | 4.690630  | 0.938126  | -4.682017  | -0.936403 | -2.717209 | -0.543442 |
| 2000-01-06 | 3.880630  | 0.646772  | -4.447700  | -0.741283 | -1.078947 | -0.179825 |
| 2000-01-07 | 4.001957  | 0.571708  | -2.884072  | -0.412010 | -3.116903 | -0.445272 |
| ...        | ...       | ...       | ...        | ...       | ...       | ...       |
| 2002-09-20 | 2.652493  | 0.044208  | -10.528875 | -0.175481 | 9.867805  | 0.164463  |
| 2002-09-21 | 0.844497  | 0.014075  | -9.280944  | -0.154682 | 9.522649  | 0.158711  |
| 2002-09-22 | 2.860036  | 0.047667  | -9.270337  | -0.154506 | 6.415245  | 0.106921  |
| 2002-09-23 | 3.510163  | 0.058503  | -8.151439  | -0.135857 | 5.177219  | 0.086287  |
| 2002-09-24 | 6.524983  | 0.108750  | -10.168078 | -0.169468 | 5.792639  | 0.096544  |
| 2002-09-25 | 6.409626  | 0.106827  | -9.956226  | -0.165937 | 5.704050  | 0.095068  |
| 2002-09-26 | 5.093787  | 0.084896  | -7.074515  | -0.117909 | 6.905823  | 0.115097  |

```
[1000 rows x 6 columns]
```

Passing a dict of functions has different behavior by default, see the next section.

### 14.3.2 Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a `DataFrame`:

```
In [93]: r.agg({'A' : np.sum,
.....:         'B' : lambda x: np.std(x, ddof=1)})
.....:
Out [93]:
```

|            | A         | B        |
|------------|-----------|----------|
| 2000-01-01 | -0.289838 | NaN      |
| 2000-01-02 | -0.216612 | 0.660747 |
| 2000-01-03 | 1.154661  | 0.689929 |
| 2000-01-04 | 2.969393  | 1.072199 |
| 2000-01-05 | 4.690630  | 0.939657 |
| 2000-01-06 | 3.880630  | 0.966848 |
| 2000-01-07 | 4.001957  | 1.240137 |
| ...        | ...       | ...      |
| 2002-09-20 | 2.652493  | 1.114814 |
| 2002-09-21 | 0.844497  | 1.113220 |
| 2002-09-22 | 2.860036  | 1.113208 |
| 2002-09-23 | 3.510163  | 1.132381 |
| 2002-09-24 | 6.524983  | 1.080963 |
| 2002-09-25 | 6.409626  | 1.082911 |
| 2002-09-26 | 5.093787  | 1.136199 |

[1000 rows x 2 columns]

The function names can also be strings. In order for a string to be valid it must be implemented on the windowed object

```
In [94]: r.agg({'A' : 'sum', 'B' : 'std'})
Out [94]:
```

|            | A         | B        |
|------------|-----------|----------|
| 2000-01-01 | -0.289838 | NaN      |
| 2000-01-02 | -0.216612 | 0.660747 |
| 2000-01-03 | 1.154661  | 0.689929 |
| 2000-01-04 | 2.969393  | 1.072199 |
| 2000-01-05 | 4.690630  | 0.939657 |
| 2000-01-06 | 3.880630  | 0.966848 |
| 2000-01-07 | 4.001957  | 1.240137 |
| ...        | ...       | ...      |
| 2002-09-20 | 2.652493  | 1.114814 |
| 2002-09-21 | 0.844497  | 1.113220 |
| 2002-09-22 | 2.860036  | 1.113208 |
| 2002-09-23 | 3.510163  | 1.132381 |
| 2002-09-24 | 6.524983  | 1.080963 |
| 2002-09-25 | 6.409626  | 1.082911 |
| 2002-09-26 | 5.093787  | 1.136199 |

[1000 rows x 2 columns]

Furthermore you can pass a nested dict to indicate different aggregations on different columns.

```
In [95]: r.agg({'A' : ['sum', 'std'], 'B' : ['mean', 'std'] })
Out [95]:
```

|            | A         |     | B         |     |
|------------|-----------|-----|-----------|-----|
|            | sum       | std | mean      | std |
| 2000-01-01 | -0.289838 | NaN | -0.370545 | NaN |

(continues on next page)



```
[1000 rows x 4 columns]
```

### 14.4.1 Method Summary

| Function                | Description                     |
|-------------------------|---------------------------------|
| <code>count()</code>    | Number of non-null observations |
| <code>sum()</code>      | Sum of values                   |
| <code>mean()</code>     | Mean of values                  |
| <code>median()</code>   | Arithmetic median of values     |
| <code>min()</code>      | Minimum                         |
| <code>max()</code>      | Maximum                         |
| <code>std()</code>      | Unbiased standard deviation     |
| <code>var()</code>      | Unbiased variance               |
| <code>skew()</code>     | Unbiased skewness (3rd moment)  |
| <code>kurt()</code>     | Unbiased kurtosis (4th moment)  |
| <code>quantile()</code> | Sample quantile (value at %)    |
| <code>apply()</code>    | Generic apply                   |
| <code>cov()</code>      | Unbiased covariance (binary)    |
| <code>corr()</code>     | Correlation (binary)            |

Aside from not having a window parameter, these functions have the same interfaces as their `.rolling` counterparts. Like above, the parameters they all accept are:

- `min_periods`: threshold of non-null data points to require. Defaults to minimum needed to compute statistic. No NaNs will be output once `min_periods` non-null data points have been seen.
- `center`: boolean, whether to set the labels at the center (default is False).

**Note:** The output of the `.rolling` and `.expanding` methods do not return a NaN if there are at least `min_periods` non-null values in the current window. For example:

```
In [98]: sn = pd.Series([1, 2, np.nan, 3, np.nan, 4])

In [99]: sn
Out[99]:
0    1.0
1    2.0
2    NaN
3    3.0
4    NaN
5    4.0
dtype: float64

In [100]: sn.rolling(2).max()
Out[100]:
0    NaN
1    2.0
2    NaN
3    NaN
4    NaN
5    NaN
dtype: float64

In [101]: sn.rolling(2, min_periods=1).max()
Out[101]:
0    1.0
1    2.0
2    3.0
3    3.0
4    4.0
5    4.0
dtype: float64
```

(continues on next page)

(continued from previous page)

```

0    1.0
1    2.0
2    2.0
3    3.0
4    3.0
5    4.0
dtype: float64

```

In case of expanding functions, this differs from `cumsum()`, `cumprod()`, `cummax()`, and `cummin()`, which return NaN in the output wherever a NaN is encountered in the input. In order to match the output of `cumsum` with expanding, use `fillna()`:

```
In [102]: sn.expanding().sum()
```

```

Out[102]:
0    1.0
1    3.0
2    3.0
3    6.0
4    6.0
5   10.0
dtype: float64

```

```
In [103]: sn.cumsum()
```

```

Out[103]:
0    1.0
1    3.0
2    NaN
3    6.0
4    NaN
5   10.0
dtype: float64

```

```
In [104]: sn.cumsum().fillna(method='ffill')
```

```

Out[104]:
0    1.0
1    3.0
2    3.0
3    6.0
4    6.0
5   10.0
dtype: float64

```

An expanding window statistic will be more stable (and less responsive) than its rolling window counterpart as the increasing window size decreases the relative impact of an individual data point. As an example, here is the `mean()` output for the previous time series dataset:

```
In [105]: s.plot(style='k--')
```

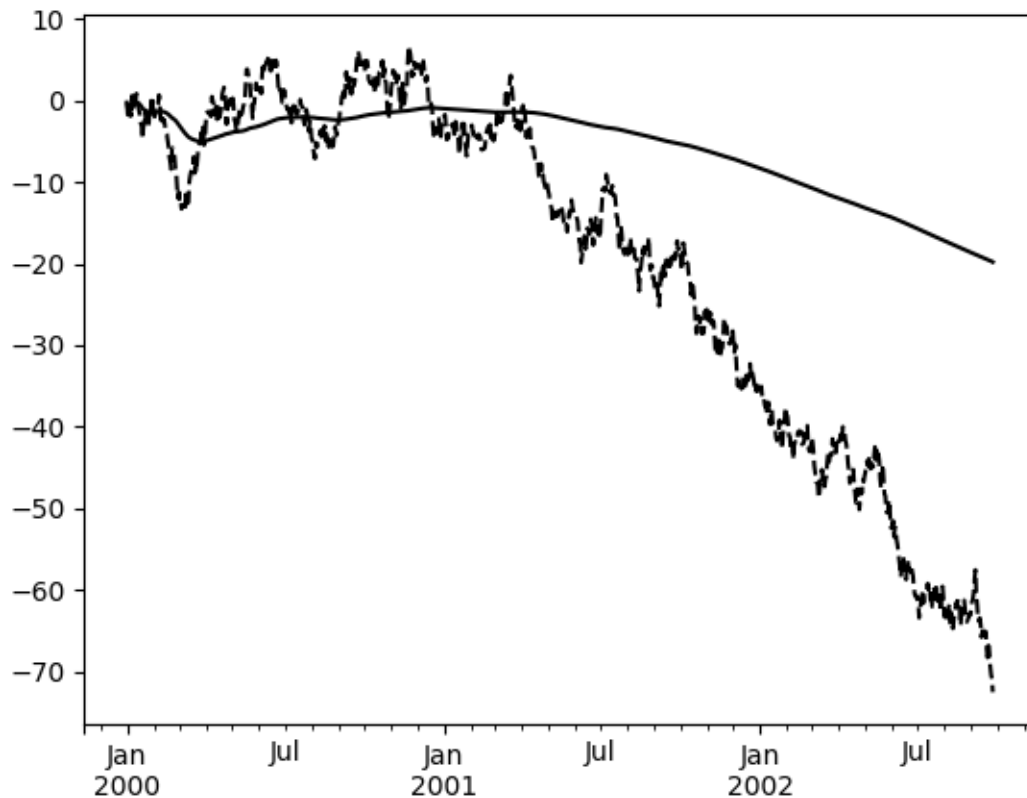
```
Out[105]: <matplotlib.axes._subplots.AxesSubplot at 0x1c254345f8>
```

```
In [106]: s.expanding().mean().plot(style='k')
```

```

Out[106]:
<matplotlib.axes._subplots.AxesSubplot at 0x1c254345f8>

```



## 14.5 Exponentially Weighted Windows

A related set of functions are exponentially weighted versions of several of the above statistics. A similar interface to `.rolling` and `.expanding` is accessed through the `.ewm` method to receive an EWM object. A number of expanding EW (exponentially weighted) methods are provided:

| Function            | Description                  |
|---------------------|------------------------------|
| <code>mean()</code> | EW moving average            |
| <code>var()</code>  | EW moving variance           |
| <code>std()</code>  | EW moving standard deviation |
| <code>corr()</code> | EW moving correlation        |
| <code>cov()</code>  | EW moving covariance         |

In general, a weighted moving average is calculated as

$$y_t = \frac{\sum_{i=0}^t w_i x_{t-i}}{\sum_{i=0}^t w_i},$$

where  $x_t$  is the input,  $y_t$  is the result and the  $w_i$  are the weights.

The EW functions support two variants of exponential weights. The default, `adjust=True`, uses the weights  $w_i =$

$(1 - \alpha)^i$  which gives

$$y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \dots + (1 - \alpha)^tx_0}{1 + (1 - \alpha) + (1 - \alpha)^2 + \dots + (1 - \alpha)^t}$$

When `adjust=False` is specified, moving averages are calculated as

$$\begin{aligned} y_0 &= x_0 \\ y_t &= (1 - \alpha)y_{t-1} + \alpha x_t, \end{aligned}$$

which is equivalent to using weights

$$w_i = \begin{cases} \alpha(1 - \alpha)^i & \text{if } i < t \\ (1 - \alpha)^i & \text{if } i = t. \end{cases}$$

---

**Note:** These equations are sometimes written in terms of  $\alpha' = 1 - \alpha$ , e.g.

$$y_t = \alpha' y_{t-1} + (1 - \alpha')x_t.$$


---

The difference between the above two variants arises because we are dealing with series which have finite history. Consider a series of infinite history:

$$y_t = \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \dots}{1 + (1 - \alpha) + (1 - \alpha)^2 + \dots}$$

Noting that the denominator is a geometric series with initial term equal to 1 and a ratio of  $1 - \alpha$  we have

$$\begin{aligned} y_t &= \frac{x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \dots}{\frac{1}{1 - (1 - \alpha)}} \\ &= [x_t + (1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \dots]\alpha \\ &= \alpha x_t + [(1 - \alpha)x_{t-1} + (1 - \alpha)^2x_{t-2} + \dots]\alpha \\ &= \alpha x_t + (1 - \alpha)[x_{t-1} + (1 - \alpha)x_{t-2} + \dots]\alpha \\ &= \alpha x_t + (1 - \alpha)y_{t-1} \end{aligned}$$

which shows the equivalence of the above two variants for infinite series. When `adjust=True` we have  $y_0 = x_0$  and from the last representation above we have  $y_t = \alpha x_t + (1 - \alpha)y_{t-1}$ , therefore there is an assumption that  $x_0$  is not an ordinary value but rather an exponentially weighted moment of the infinite series up to that point.

One must have  $0 < \alpha \leq 1$ , and while since version 0.18.0 it has been possible to pass  $\alpha$  directly, it's often easier to think about either the **span**, **center of mass (com)** or **half-life** of an EW moment:

$$\alpha = \begin{cases} \frac{2}{s+1}, & \text{for span } s \geq 1 \\ \frac{1}{1+c}, & \text{for center of mass } c \geq 0 \\ 1 - \exp^{-\frac{\log 0.5}{h}}, & \text{for half-life } h > 0 \end{cases}$$

One must specify precisely one of **span**, **center of mass**, **half-life** and **alpha** to the EW functions:

- **Span** corresponds to what is commonly called an “N-day EW moving average”.
- **Center of mass** has a more physical interpretation and can be thought of in terms of span:  $c = (s - 1)/2$ .
- **Half-life** is the period of time for the exponential weight to reduce to one half.
- **Alpha** specifies the smoothing factor directly.

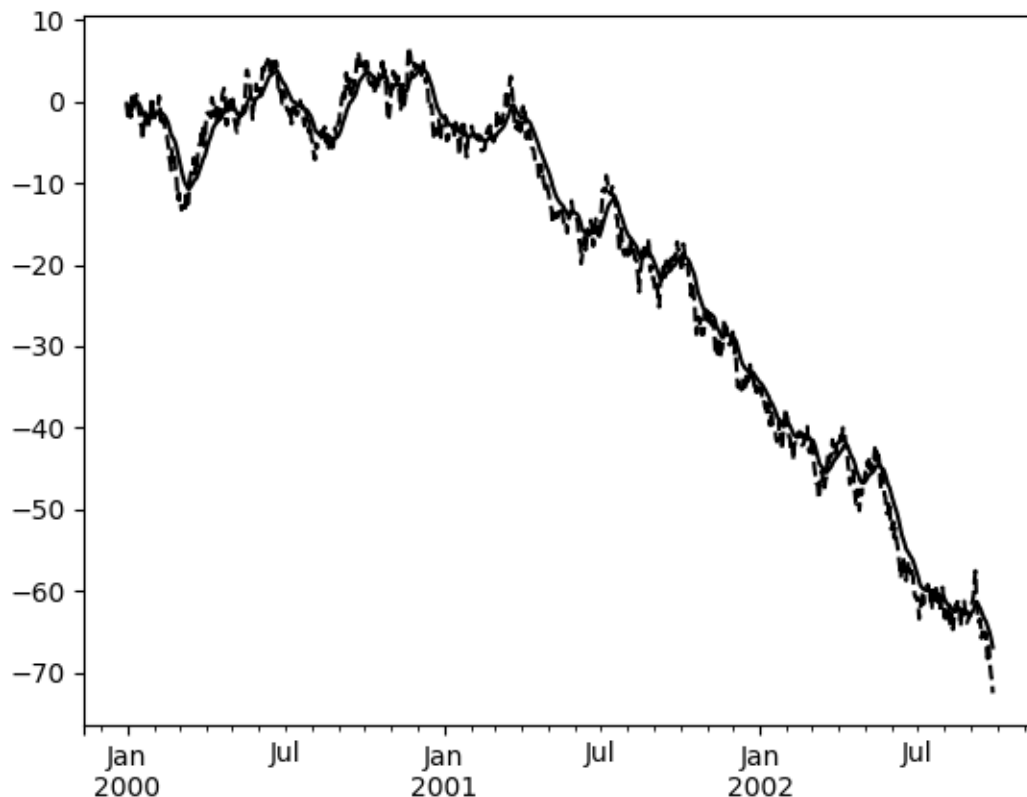
Here is an example for a univariate time series:

```

In [107]: s.plot(style='k--')
Out[107]: <matplotlib.axes._subplots.AxesSubplot at 0x1a2490e630>

In [108]: s.ewm(span=20).mean().plot(style='k')
Out[108]:
\\Out[108]:
↪<matplotlib.axes._subplots.AxesSubplot at 0x1a2490e630>

```



EWM has a `min_periods` argument, which has the same meaning it does for all the `.expanding` and `.rolling` methods: no output values will be set until at least `min_periods` non-null values are encountered in the (expanding) window.

EWM also has an `ignore_na` argument, which determines how intermediate null values affect the calculation of the weights. When `ignore_na=False` (the default), weights are calculated based on absolute positions, so that intermediate null values affect the result. When `ignore_na=True`, weights are calculated by ignoring intermediate null values. For example, assuming `adjust=True`, if `ignore_na=False`, the weighted average of 3, NaN, 5 would be calculated as

$$\frac{(1 - \alpha)^2 \cdot 3 + 1 \cdot 5}{(1 - \alpha)^2 + 1}.$$

Whereas if `ignore_na=True`, the weighted average would be calculated as

$$\frac{(1 - \alpha) \cdot 3 + 1 \cdot 5}{(1 - \alpha) + 1}.$$

The `var()`, `std()`, and `cov()` functions have a `bias` argument, specifying whether the result should contain biased or unbiased statistics. For example, if `bias=True`, `ewmvar(x)` is calculated as `ewmvar(x) =`

`ewma(x**2) - ewma(x)**2`; whereas if `bias=False` (the default), the biased variance statistics are scaled by debiasing factors

$$\frac{\left(\sum_{i=0}^t w_i\right)^2}{\left(\sum_{i=0}^t w_i\right)^2 - \sum_{i=0}^t w_i^2}.$$

(For  $w_i = 1$ , this reduces to the usual  $N/(N - 1)$  factor, with  $N = t + 1$ .) See [Weighted Sample Variance](#) on Wikipedia for further details.





## WORKING WITH MISSING DATA

In this section, we will discuss missing (also referred to as NA) values in pandas.

---

**Note:** The choice of using NaN internally to denote missing data was largely for simplicity and performance reasons. It differs from the MaskedArray approach of, for example, `scikits.timeseries`. We are hopeful that NumPy will soon be able to provide a native NA type solution (similar to R) performant enough to be used in pandas.

---

See the *cookbook* for some advanced strategies.

### 15.1 Missing data basics

#### 15.1.1 When / why does data become missing?

Some might quibble over our usage of *missing*. By “missing” we simply mean **NA** (“not available”) or “not present for whatever reason”. Many data sets simply arrive with missing data, either because it exists and was not collected or it never existed. For example, in a collection of financial time series, some of the time series might start on different dates. Thus, values prior to the start date would generally be marked as missing.

In pandas, one of the most common ways that missing data is **introduced** into a data set is by reindexing. For example:

```
In [1]: df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f', 'h'],
...:                      columns=['one', 'two', 'three'])
...:
...:

In [2]: df['four'] = 'bar'

In [3]: df['five'] = df['one'] > 0

In [4]: df
Out[4]:
```

|   | one       | two       | three     | four | five  |
|---|-----------|-----------|-----------|------|-------|
| a | -0.166778 | 0.501113  | -0.355322 | bar  | False |
| c | -0.337890 | 0.580967  | 0.983801  | bar  | False |
| e | 0.057802  | 0.761948  | -0.712964 | bar  | True  |
| f | -0.443160 | -0.974602 | 1.047704  | bar  | False |
| h | -0.717852 | -1.053898 | -0.019369 | bar  | False |

```
In [5]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [6]: df2
Out[6]:
```

(continues on next page)

(continued from previous page)

|   | one       | two       | three     | four | five  |
|---|-----------|-----------|-----------|------|-------|
| a | -0.166778 | 0.501113  | -0.355322 | bar  | False |
| b | NaN       | NaN       | NaN       | NaN  | NaN   |
| c | -0.337890 | 0.580967  | 0.983801  | bar  | False |
| d | NaN       | NaN       | NaN       | NaN  | NaN   |
| e | 0.057802  | 0.761948  | -0.712964 | bar  | True  |
| f | -0.443160 | -0.974602 | 1.047704  | bar  | False |
| g | NaN       | NaN       | NaN       | NaN  | NaN   |
| h | -0.717852 | -1.053898 | -0.019369 | bar  | False |

### 15.1.2 Values considered “missing”

As data comes in many shapes and forms, pandas aims to be flexible with regard to handling missing data. While NaN is the default missing value marker for reasons of computational speed and convenience, we need to be able to easily detect this value with data of different types: floating point, integer, boolean, and general object. In many cases, however, the Python None will arise and we wish to also consider that “missing” or “not available” or “NA”.

---

**Note:** If you want to consider inf and -inf to be “NA” in computations, you can set `pandas.options.mode.use_inf_as_na = True`.

---

To make detecting missing values easier (and across different array dtypes), pandas provides the `isna()` and `notna()` functions, which are also methods on Series and DataFrame objects:

```
In [7]: df2['one']
Out[7]:
a    -0.166778
b         NaN
c    -0.337890
d         NaN
e     0.057802
f    -0.443160
g         NaN
h    -0.717852
Name: one, dtype: float64

In [8]: pd.isna(df2['one'])
Out[8]:
a    False
b     True
c    False
d     True
e    False
f    False
g     True
h    False
Name: one, dtype: bool

In [9]: df2['four'].notna()
Out[9]:
a     True
b    False
```

(continues on next page)

```
c      True
d      False
e      True
f      True
g      False
h      True
Name: four, dtype: bool
```

```

      one      two  three   four   five
a  False  False  False  False  False
b   True   True   True   True   True
c  False  False  False  False  False
d   True   True   True   True   True
e  False  False  False  False  False
f  False  False  False  False  False
g   True   True   True   True   True
h  False  False  False  False  False

```

### 15.3 Inserting missing data

You can insert missing values by simply assigning to containers. The actual missing value used will be chosen based on the dtype.

For example, numeric containers will always use NaN regardless of the missing value type chosen:

```
In [20]: s = pd.Series([1, 2, 3])

In [21]: s.loc[0] = None

In [22]: s
Out[22]:
0      NaN
1      2.0
2      3.0
dtype: float64
```

Likewise, datetime containers will always use NaT.

For object containers, pandas will use the value given:

```
In [23]: s = pd.Series(["a", "b", "c"])
In [24]: s.loc[0] = None
```

(continues on next page)

(continued from previous page)

```
In [25]: s.loc[1] = np.nan

In [26]: s
Out[26]:
0      None
1      NaN
2         c
dtype: object
```

## 15.4 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

```
In [27]: a
Out[27]:
      one      two
a      NaN  0.501113
c      NaN  0.580967
e  0.057802  0.761948
f -0.443160 -0.974602
h -0.443160 -1.053898
```

```
In [28]: b
↗
      one      two      three
a      NaN  0.501113 -0.355322
c      NaN  0.580967  0.983801
e  0.057802  0.761948 -0.712964
f -0.443160 -0.974602  1.047704
h      NaN -1.053898 -0.019369
```

```
In [29]: a + b
↗
      one  three      two
a      NaN  NaN  1.002226
c      NaN  NaN  1.161935
e  0.115604  NaN  1.523896
f -0.886321  NaN -1.949205
h      NaN  NaN -2.107796
```

The descriptive statistics and computational methods discussed in the [data structure overview](#) (and listed [here](#) and [here](#)) are all written to account for missing data. For example:

- When summing data, NA (missing) values will be treated as zero.
- If the data are all NA, the result will be 0.
- Cumulative methods like `cumsum()` and `cumprod()` ignore NA values by default, but preserve them in the resulting arrays. To override this behaviour and include NA values, use `skipna=False`.

```
In [30]: df
Out[30]:
```

(continues on next page)

(continued from previous page)

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | NaN       | 0.501113  | -0.355322 |
| c | NaN       | 0.580967  | 0.983801  |
| e | 0.057802  | 0.761948  | -0.712964 |
| f | -0.443160 | -0.974602 | 1.047704  |
| h | NaN       | -1.053898 | -0.019369 |

```
In [31]: df['one'].sum()
```

$\frac{1}{\sqrt{e}}$

```
In [32]: df.mean(1)
```

```
In [33]: df.cumsum()
```

```
In [34]: df.cumsum(skipna=False)
```

|   | one | two       | three     |
|---|-----|-----------|-----------|
| a | NaN | 0.501113  | -0.355322 |
| c | NaN | 1.082080  | 0.628479  |
| e | NaN | 1.844028  | -0.084485 |
| f | NaN | 0.869426  | 0.963219  |
| h | NaN | -0.184472 | 0.943850  |

### 15.4.1 Sum/Prod of Empties/Nans

**Warning:** This behavior is now standard as of v0.22.0 and is consistent with the default in `numpy`; previously `sum/prod` of all-NA or empty `Series/DataFrames` would return `NaN`. See [v0.22.0 whatsnew](#) for more.

The sum of an empty or all-NA Series or column of a DataFrame is 0.

```
In [35]: pd.Series([np.nan]).sum()
```

Out [35]: 0.0

```
In [36]: pd.Series([]).sum()
```

```
Out[36]: 0.0
```

The product of an empty or all-NA Series or column of a DataFrame is 1.

```
In [37]: pd.Series([np.nan]).prod()
Out[37]: 1.0

In [38]: pd.Series([]).prod()
Out[38]: 1.0
```

## 15.4.2 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example:

```
In [39]: df
Out[39]:
```

|   | one       | two       | three     |
|---|-----------|-----------|-----------|
| a | NaN       | 0.501113  | -0.355322 |
| c | NaN       | 0.580967  | 0.983801  |
| e | 0.057802  | 0.761948  | -0.712964 |
| f | -0.443160 | -0.974602 | 1.047704  |
| h | NaN       | -1.053898 | -0.019369 |

```
In [40]: df.groupby('one').mean()
Out[40]:
```

|           | two       | three     |
|-----------|-----------|-----------|
| one       |           |           |
| -0.443160 | -0.974602 | 1.047704  |
| 0.057802  | 0.761948  | -0.712964 |

See the groupby section [here](#) for more information.

## 15.5 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

### 15.5.1 Filling missing values: fillna

`fillna()` can “fill in” NA values with non-NA data in a couple of ways, which we illustrate:

**Replace NA with a scalar value**

```
In [41]: df2
Out[41]:
```

|   | one       | two       | three     | four | five  | timestamp  |
|---|-----------|-----------|-----------|------|-------|------------|
| a | NaN       | 0.501113  | -0.355322 | bar  | False | NaT        |
| c | NaN       | 0.580967  | 0.983801  | bar  | False | NaT        |
| e | 0.057802  | 0.761948  | -0.712964 | bar  | True  | 2012-01-01 |
| f | -0.443160 | -0.974602 | 1.047704  | bar  | False | 2012-01-01 |
| h | NaN       | -1.053898 | -0.019369 | bar  | False | NaT        |

```
In [42]: df2.fillna(0)
```

(continues on next page)

(continued from previous page)

```

      one      two      three four   five      timestamp
a  0.000000  0.501113 -0.355322  bar  False              0
c  0.000000  0.580967  0.983801  bar  False              0
e  0.057802  0.761948 -0.712964  bar   True  2012-01-01 00:00:00
f -0.443160 -0.974602  1.047704  bar  False  2012-01-01 00:00:00
h  0.000000 -1.053898 -0.019369  bar  False              0

```

```
In [43]: df2['one'].fillna('missing')
```

```

////////////////////////////////////
↪
a      missing
c      missing
e      0.057802
f     -0.44316
h      missing
Name: one, dtype: object

```

### Fill gaps forward or backward

Using the same filling arguments as [reindexing](#), we can propagate non-NA values forward or backward:

```
In [44]: df
```

```
Out [44]:
```

```

      one      two      three
a      NaN  0.501113 -0.355322
c      NaN  0.580967  0.983801
e  0.057802  0.761948 -0.712964
f -0.443160 -0.974602  1.047704
h      NaN -1.053898 -0.019369

```

```
In [45]: df.fillna(method='pad')
```

```

////////////////////////////////////
↪
      one      two      three
a      NaN  0.501113 -0.355322
c      NaN  0.580967  0.983801
e  0.057802  0.761948 -0.712964
f -0.443160 -0.974602  1.047704
h -0.443160 -1.053898 -0.019369

```

### Limit the amount of filling

If we only want consecutive gaps filled up to a certain number of data points, we can use the *limit* keyword:

```
In [46]: df
```

```
Out [46]:
```

```

      one      two      three
a  NaN  0.501113 -0.355322
c  NaN  0.580967  0.983801
e  NaN      NaN      NaN
f  NaN      NaN      NaN
h  NaN -1.053898 -0.019369

```

```
In [47]: df.fillna(method='pad', limit=1)
```

```

////////////////////////////////////
↪
      one      two      three

```

(continues on next page)



(continued from previous page)

```

a  NaN    0.501113 -0.355322
c  NaN    0.580967  0.983801
e  NaN    0.580967  0.983801
f  NaN          NaN          NaN
h  NaN   -1.053898 -0.019369

```

To remind you, these are the available filling methods:

| Method           | Action               |
|------------------|----------------------|
| pad / ffill      | Fill values forward  |
| bfill / backfill | Fill values backward |

With time series data, using pad/ffill is extremely common so that the “last known value” is available at every time point.

`ffill()` is equivalent to `fillna(method='ffill')` and `bfill()` is equivalent to `fillna(method='bfill')`

## 15.5.2 Filling with a PandasObject

You can also fillna using a dict or Series that is alignable. The labels of the dict or index of the Series must match the columns of the frame you wish to fill. The use case of this is to fill a DataFrame with the mean of that column.

```
In [48]: dff = pd.DataFrame(np.random.randn(10,3), columns=list('ABC'))
```

```
In [49]: dff.iloc[3:5,0] = np.nan
```

```
In [50]: dff.iloc[4:6,1] = np.nan
```

```
In [51]: dff.iloc[5:8,2] = np.nan
```

```
In [52]: dff
```

```
Out[52]:
```

```

      A         B         C
0  0.758887  2.340598  0.219039
1 -1.235583  0.031785  0.701683
2 -1.557016 -0.636986 -1.238610
3      NaN -1.002278  0.654052
4      NaN      NaN  1.053999
5  0.651981      NaN      NaN
6  0.109001 -0.533294      NaN
7 -1.037831 -1.150016      NaN
8 -0.687693  1.921056 -0.121113
9 -0.258742 -0.706329  0.402547

```

```
In [53]: dff.fillna(dff.mean())
```

```

////////////////////////////////////

```

```

      A         B         C
0  0.758887  2.340598  0.219039
1 -1.235583  0.031785  0.701683
2 -1.557016 -0.636986 -1.238610
3 -0.407125 -1.002278  0.654052
4 -0.407125  0.033067  1.053999

```

(continues on next page)

(continued from previous page)

|   |           |           |           |
|---|-----------|-----------|-----------|
| 5 | 0.651981  | 0.033067  | 0.238800  |
| 6 | 0.109001  | -0.533294 | 0.238800  |
| 7 | -1.037831 | -1.150016 | 0.238800  |
| 8 | -0.687693 | 1.921056  | -0.121113 |
| 9 | -0.258742 | -0.706329 | 0.402547  |

```
In [54]: dff.fillna(dff.mean()['B':'C'])
```

|   | A         | B         | C         |
|---|-----------|-----------|-----------|
| 0 | 0.758887  | 2.340598  | 0.219039  |
| 1 | -1.235583 | 0.031785  | 0.701683  |
| 2 | -1.557016 | -0.636986 | -1.238610 |
| 3 | NaN       | -1.002278 | 0.654052  |
| 4 | NaN       | 0.033067  | 1.053999  |
| 5 | 0.651981  | 0.033067  | 0.238800  |
| 6 | 0.109001  | -0.533294 | 0.238800  |
| 7 | -1.037831 | -1.150016 | 0.238800  |
| 8 | -0.687693 | 1.921056  | -0.121113 |
| 9 | -0.258742 | -0.706329 | 0.402547  |

Same result as above, but is aligning the ‘fill’ value which is a Series in this case.

```
In [55]: dff.where(pd.notna(dff), dff.mean(), axis='columns')
```

Out [55] :

|   | A         | B         | C         |
|---|-----------|-----------|-----------|
| 0 | 0.758887  | 2.340598  | 0.219039  |
| 1 | -1.235583 | 0.031785  | 0.701683  |
| 2 | -1.557016 | -0.636986 | -1.238610 |
| 3 | -0.407125 | -1.002278 | 0.654052  |
| 4 | -0.407125 | 0.033067  | 1.053999  |
| 5 | 0.651981  | 0.033067  | 0.238800  |
| 6 | 0.109001  | -0.533294 | 0.238800  |
| 7 | -1.037831 | -1.150016 | 0.238800  |
| 8 | -0.687693 | 1.921056  | -0.121113 |
| 9 | -0.258742 | -0.706329 | 0.402547  |

### 15.5.3 Dropping axis labels with missing data: dropna

You may wish to simply exclude labels from a data set which refer to missing data. To do this, use `dropna()`:

```
In [56]: df
```

Out [56] :

|   | one | two       | three     |
|---|-----|-----------|-----------|
| a | NaN | 0.501113  | -0.355322 |
| c | NaN | 0.580967  | 0.983801  |
| e | NaN | 0.000000  | 0.000000  |
| f | NaN | 0.000000  | 0.000000  |
| h | NaN | -1.053898 | -0.019369 |

```
In [57]: df.dropna(axis=0)
```

Empty DataFrame

```
Columns: [one, two, three]
```

(continues on next page)

(continued from previous page)

Index: []

```
In [58]: df.dropna(axis=1)
```

|   | two       | three     |
|---|-----------|-----------|
| a | 0.501113  | -0.355322 |
| c | 0.580967  | 0.983801  |
| e | 0.000000  | 0.000000  |
| f | 0.000000  | 0.000000  |
| h | -1.053898 | -0.019369 |

```
In [59]: df['one'].dropna()
```

```
→ Series([], Name: one, dtype: float64)
```

An equivalent `dropna()` is available for Series. `DataFrame.dropna` has considerably more options than `Series.dropna`, which can be examined *in the API*.

### 15.5.4 Interpolation

New in version 0.21.0: The `limit_area` keyword argument was added.

Both Series and DataFrame objects have `interpolate()` that, by default, performs linear interpolation at missing datapoints.

```
In [60]: ts
```

Out [60]:

|            |          |
|------------|----------|
| 2000-01-31 | 0.469112 |
| 2000-02-29 | NaN      |
| 2000-03-31 | NaN      |
| 2000-04-28 | NaN      |
| 2000-05-31 | NaN      |
| 2000-06-30 | NaN      |
| 2000-07-31 | NaN      |

|            |           |
|------------|-----------|
|            | ...       |
| 2007-10-31 | -3.305259 |
| 2007-11-30 | -5.485119 |
| 2007-12-31 | -6.854968 |
| 2008-01-31 | -7.809176 |
| 2008-02-29 | -6.346480 |
| 2008-03-31 | -8.089641 |
| 2008-04-30 | -8.916232 |

Freq: BM, Length: 100, dtype: float64

```
In [61]: ts.count()
```

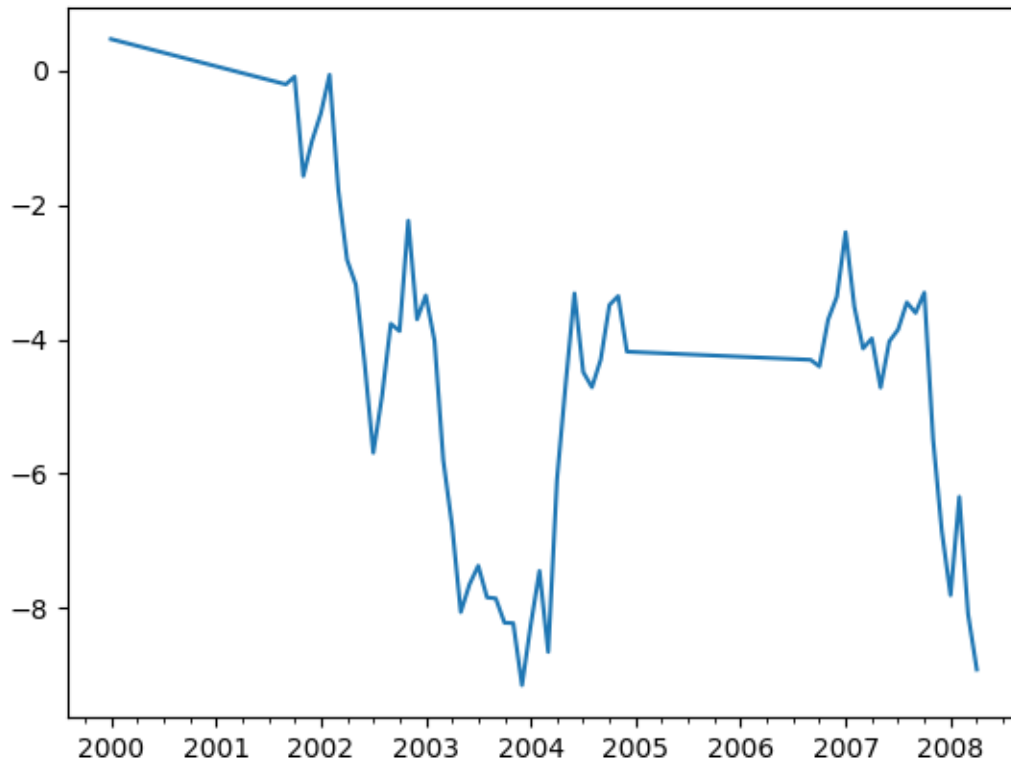
↪ 61

```
In [62]: ts.interpolate().count()
```

→ 100

```
In [63]: ts.interpolate().plot()
```

```
→<matplotlib.axes._subplots.AxesSubplot at 0x1c38e4a048>
```



Index aware interpolation is available via the `method` keyword:

```
In [64]: ts2
```

Out [64] :

|            |          |
|------------|----------|
| 2000-01-31 | 0.469112 |
|------------|----------|

|            |     |
|------------|-----|
| 2000-02-29 | NaN |
|------------|-----|

|            |           |
|------------|-----------|
| 2002-07-31 | -5.689738 |
|------------|-----------|

|            |     |
|------------|-----|
| 2005-01-31 | NaN |
|------------|-----|

|            |           |
|------------|-----------|
| 2008-04-30 | -8.916232 |
|------------|-----------|

```
dtype: float64
```

```
In [65]: ts2.interpolate()
```



|            |          |
|------------|----------|
| 2000-01-31 | 0.469112 |
|------------|----------|

|            |           |
|------------|-----------|
| 2000-02-29 | -2.610313 |
|------------|-----------|

|            |           |
|------------|-----------|
| 2002-07-31 | -5.689738 |
|------------|-----------|

|            |           |
|------------|-----------|
| 2005-01-31 | -7.302985 |
|------------|-----------|

|            |           |
|------------|-----------|
| 2008-04-30 | -8.916232 |
|------------|-----------|

```
dtype: float64
```

```
In [66]: ts2.interpolate(method='time')
```

////////////////////////////////////



|            |          |
|------------|----------|
| 2000-01-31 | 0.469112 |
|------------|----------|

|            |          |
|------------|----------|
| 2000-01-31 | 0.189112 |
| 2000-02-29 | 0.273272 |

(continues on next page)

(continued from previous page)

```

2002-07-31    -5.689738
2005-01-31    -7.095568
2008-04-30    -8.916232
dtype: float64

```

For a floating-point index, use `method='values'`:

```

In [67]: ser
Out[67]:
0.0      0.0
1.0      NaN
10.0     10.0
dtype: float64

In [68]: ser.interpolate()
Out[68]:
0.0      0.0
1.0      5.0
10.0     10.0
dtype: float64

In [69]: ser.interpolate(method='values')
Out[69]:
0.0      0.0
1.0      1.0
10.0     10.0
dtype: float64

```

You can also interpolate with a DataFrame:

```

In [70]: df = pd.DataFrame({'A': [1, 2.1, np.nan, 4.7, 5.6, 6.8],
.....:                    'B': [.25, np.nan, np.nan, 4, 12.2, 14.4]})

In [71]: df
Out[71]:
   A      B
0  1.0  0.25
1  2.1   NaN
2  NaN   NaN
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40

In [72]: df.interpolate()
Out[72]:
   A      B
0  1.0  0.25
1  2.1  1.50
2  3.4  2.75
3  4.7  4.00
4  5.6 12.20
5  6.8 14.40

```

The `method` argument gives access to fancier interpolation methods. If you have `scipy` installed, you can pass the

name of a 1-d interpolation routine to `method`. You'll want to consult the full [scipy interpolation documentation](#) and reference [guide](#) for details. The appropriate interpolation method will depend on the type of data you are working with.

- If you are dealing with a time series that is growing at an increasing rate, `method='quadratic'` may be appropriate.
- If you have values approximating a cumulative distribution function, then `method='pchip'` should work well.
- To fill missing values with goal of smooth plotting, consider `method='akima'`.

**Warning:** These methods require `scipy`.

```
In [73]: df.interpolate(method='barycentric')
```

```
Out[73]:
```

|   | A    | B      |
|---|------|--------|
| 0 | 1.00 | 0.250  |
| 1 | 2.10 | -7.660 |
| 2 | 3.53 | -4.515 |
| 3 | 4.70 | 4.000  |
| 4 | 5.60 | 12.200 |
| 5 | 6.80 | 14.400 |

```
In [74]: df.interpolate(method='pchip')
```

```
////////////////////////////////////
```

```
↪
```

|   | A        | B         |
|---|----------|-----------|
| 0 | 1.000000 | 0.250000  |
| 1 | 2.100000 | 0.672808  |
| 2 | 3.43454  | 1.928950  |
| 3 | 4.700000 | 4.000000  |
| 4 | 5.600000 | 12.200000 |
| 5 | 6.800000 | 14.400000 |

```
In [75]: df.interpolate(method='akima')
```

```
////////////////////////////////////
```

```
↪
```

|   | A        | B         |
|---|----------|-----------|
| 0 | 1.000000 | 0.250000  |
| 1 | 2.100000 | -0.873316 |
| 2 | 3.406667 | 0.320034  |
| 3 | 4.700000 | 4.000000  |
| 4 | 5.600000 | 12.200000 |
| 5 | 6.800000 | 14.400000 |

When interpolating via a polynomial or spline approximation, you must also specify the degree or order of the approximation:

```
In [76]: df.interpolate(method='spline', order=2)
```

```
Out[76]:
```

|   | A        | B         |
|---|----------|-----------|
| 0 | 1.000000 | 0.250000  |
| 1 | 2.100000 | -0.428598 |
| 2 | 3.404545 | 1.206900  |
| 3 | 4.700000 | 4.000000  |

(continues on next page)

```
In [77]: df.interpolate(method='polynomial', order=2)
```

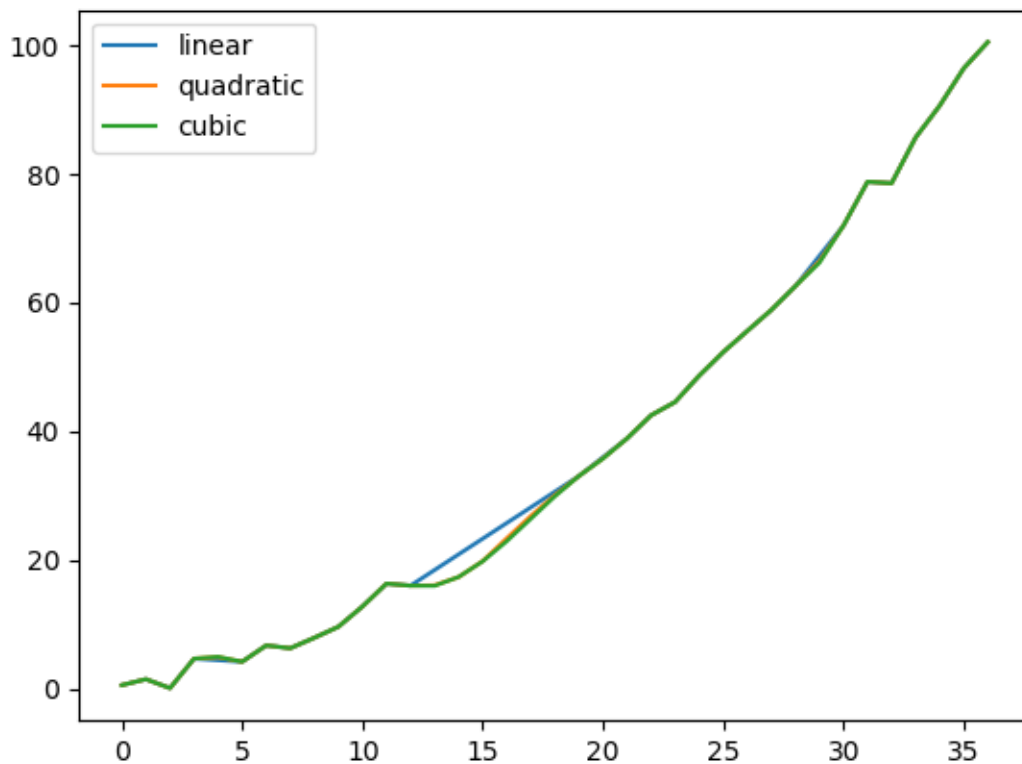


```
In [79]: ser = pd.Series(np.arange(1, 10.1, .25)**2 + np.random.randn(37))
```

```
In [81]: ser[bad] = np.nan
```

```
In [83]: df = pd.DataFrame({m: ser.interpolate(method=m) for m in methods})
```

```
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3702e518>
```



Another use case is interpolation at *new* values. Suppose you have 100 observations from some distribution. And let's suppose that you're particularly interested in what's happening around the middle. You can mix pandas' `reindex` and `interpolate` methods to interpolate at the new values.

```
In [85]: ser = pd.Series(np.sort(np.random.uniform(size=100)))

# interpolate at new_index
In [86]: new_index = ser.index | pd.Index([49.25, 49.5, 49.75, 50.25, 50.5, 50.75])

In [87]: interp_s = ser.reindex(new_index).interpolate(method='pchip')

In [88]: interp_s[49:51]
Out[88]:
49.00    0.471410
49.25    0.476841
49.50    0.481780
49.75    0.485998
50.00    0.489266
50.25    0.491814
50.50    0.493995
50.75    0.495763
51.00    0.497074
dtype: float64
```



### 15.5.4.1 Interpolation Limits

Like other pandas fill methods, `interpolate()` accepts a `limit` keyword argument. Use this argument to limit the number of consecutive NaN values filled since the last valid observation:

```
In [89]: ser = pd.Series([np.nan, np.nan, 5, np.nan, np.nan, np.nan, 13, np.nan, np.
↳ nan])

# fill all consecutive values in a forward direction
In [90]: ser.interpolate()
Out[90]:
0      NaN
1      NaN
2      5.0
3      7.0
4      9.0
5     11.0
6     13.0
7     13.0
8     13.0
dtype: float64

# fill one consecutive value in a forward direction
In [91]: ser.interpolate(limit=1)
////////////////////////////////////
↳
0      NaN
1      NaN
2      5.0
3      7.0
4      NaN
5      NaN
6     13.0
7     13.0
8      NaN
dtype: float64
```

By default, NaN values are filled in a forward direction. Use `limit_direction` parameter to fill backward or from both directions.

```
# fill one consecutive value backwards
In [92]: ser.interpolate(limit=1, limit_direction='backward')
Out[92]:
0      NaN
1      5.0
2      5.0
3      NaN
4      NaN
5     11.0
6     13.0
7      NaN
8      NaN
dtype: float64
```

```
# fill one consecutive value in both directions
In [93]: ser.interpolate(limit=1, limit_direction='both')
////////////////////////////////////
↳
```

(continues on next page)

(continued from previous page)

```

0      NaN
1      5.0
2      5.0
3      7.0
4      NaN
5      11.0
6      13.0
7      13.0
8      NaN
dtype: float64

```

```
# fill all consecutive values in both directions
```

```
In [94]: ser.interpolate(limit_direction='both')
```

```

////////////////////////////////////
↪
0      5.0
1      5.0
2      5.0
3      7.0
4      9.0
5      11.0
6      13.0
7      13.0
8      13.0
dtype: float64

```

By default, NaN values are filled whether they are inside (surrounded by) existing valid values, or outside existing valid values. Introduced in v0.23 the `limit_area` parameter restricts filling to either inside or outside values.

```
# fill one consecutive inside value in both directions
```

```
In [95]: ser.interpolate(limit_direction='both', limit_area='inside', limit=1)
```

```
Out [95]:
```

```

0      NaN
1      NaN
2      5.0
3      7.0
4      NaN
5      11.0
6      13.0
7      NaN
8      NaN
dtype: float64

```

```
# fill all consecutive outside values backward
```

```
In [96]: ser.interpolate(limit_direction='backward', limit_area='outside')
```

```

////////////////////////////////////
↪
0      5.0
1      5.0
2      5.0
3      NaN
4      NaN
5      NaN
6      13.0
7      NaN
8      NaN

```

(continues on next page)

(continued from previous page)

```
dtype: float64

# fill all consecutive outside values in both directions
In [97]: ser.interpolate(limit_direction='both', limit_area='outside')
//////////
↪
0      5.0
1      5.0
2      5.0
3      NaN
4      NaN
5      NaN
6     13.0
7     13.0
8     13.0
dtype: float64
```

### 15.5.5 Replacing Generic Values

Often times we want to replace arbitrary values with other values.

`replace()` in Series and `replace()` in DataFrame provides an efficient yet flexible way to perform such replacements.

For a Series, you can replace a single value or a list of values by another value:

```
In [98]: ser = pd.Series([0., 1., 2., 3., 4.])

In [99]: ser.replace(0, 5)
Out[99]:
0    5.0
1    1.0
2    2.0
3    3.0
4    4.0
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [100]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[100]:
0      4.0
1      3.0
2      2.0
3      1.0
4      0.0
dtype: float64
```

You can also specify a mapping dict:

```
In [101]: ser.replace({0: 10, 1: 100})
Out[101]:
0      10.0
1     100.0
2       2.0
```

(continues on next page)

(continued from previous page)

```
3      3.0
4      4.0
dtype: float64
```

For a DataFrame, you can specify individual values by column:

```
In [102]: df = pd.DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})

In [103]: df.replace({'a': 0, 'b': 5}, 100)
Out[103]:
```

|   | a   | b   |
|---|-----|-----|
| 0 | 100 | 100 |
| 1 | 1   | 6   |
| 2 | 2   | 7   |
| 3 | 3   | 8   |
| 4 | 4   | 9   |

Instead of replacing with specified values, you can treat all given values as missing and interpolate over them:

```
In [104]: ser.replace([1, 2, 3], method='pad')
Out[104]:
```

|   |     |
|---|-----|
| 0 | 0.0 |
| 1 | 0.0 |
| 2 | 0.0 |
| 3 | 0.0 |
| 4 | 4.0 |

dtype: float64

## 15.5.6 String/Regular Expression Replacement

---

**Note:** Python strings prefixed with the `r` character such as `r'hello world'` are so-called “raw” strings. They have different semantics regarding backslashes than strings without this prefix. Backslashes in raw strings will be interpreted as an escaped backslash, e.g., `r'\'` == `'\\'`. You should [read about them](#) if this is unclear.

---

Replace the `'.'` with NaN (str -> str):

```
In [105]: d = {'a': list(range(4)), 'b': list('ab..'), 'c': ['a', 'b', np.nan, 'd']}

In [106]: df = pd.DataFrame(d)

In [107]: df.replace('.', np.nan)
Out[107]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | a   | a   |
| 1 | 1 | b   | b   |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

Now do it with a regular expression that removes surrounding whitespace (regex -> regex):

```
In [108]: df.replace(r'\s*\.\s*', np.nan, regex=True)
Out[108]:
```

|  | a | b | c |
|--|---|---|---|
|--|---|---|---|

(continues on next page)

(continued from previous page)

|   |   |     |     |
|---|---|-----|-----|
| 0 | 0 | a   | a   |
| 1 | 1 | b   | b   |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

Replace a few different values (list -> list):

```
In [109]: df.replace(['a', '.'], ['b', np.nan])
```

```
Out [109]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | b   | b   |
| 1 | 1 | b   | b   |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

list of regex -> list of regex:

```
In [110]: df.replace([r'\.', r'(a)'], ['dot', '\1stuff'], regex=True)
```

```
Out [110]:
```

|   | a | b      | c      |
|---|---|--------|--------|
| 0 | 0 | {stuff | {stuff |
| 1 | 1 | b      | b      |
| 2 | 2 | dot    | NaN    |
| 3 | 3 | dot    | d      |

Only search in column 'b' (dict -> dict):

```
In [111]: df.replace({'b': '.'}, {'b': np.nan})
```

```
Out [111]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | a   | a   |
| 1 | 1 | b   | b   |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

Same as the previous example, but use a regular expression for searching instead (dict of regex -> dict):

```
In [112]: df.replace({'b': r'\s*\.\s*'}, {'b': np.nan}, regex=True)
```

```
Out [112]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | a   | a   |
| 1 | 1 | b   | b   |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

You can pass nested dictionaries of regular expressions that use `regex=True`:

```
In [113]: df.replace({'b': {'b': r'.'}}, regex=True)
```

```
Out [113]:
```

|   | a | b | c   |
|---|---|---|-----|
| 0 | 0 | a | a   |
| 1 | 1 | b | b   |
| 2 | 2 | . | NaN |
| 3 | 3 | . | d   |

Alternatively, you can pass the nested dictionary like so:

```
In [114]: df.replace(regex={'b': {r'\s*\.\s*': np.nan}})
Out[114]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | a   | a   |
| 1 | 1 | b   | b   |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

You can also use the group of a regular expression match when replacing (dict of regex -> dict of regex), this works for lists as well.

```
In [115]: df.replace({'b': r'\s*(\.)\s*'}, {'b': r'\1ty'}, regex=True)
Out[115]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | a   | a   |
| 1 | 1 | b   | b   |
| 2 | 2 | .ty | NaN |
| 3 | 3 | .ty | d   |

You can pass a list of regular expressions, of which those that match will be replaced with a scalar (list of regex -> regex).

```
In [116]: df.replace([r'\s*\.\s*', r'a|b'], np.nan, regex=True)
Out[116]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | NaN | NaN |
| 1 | 1 | NaN | NaN |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

All of the regular expression examples can also be passed with the `to_replace` argument as the `regex` argument. In this case the `value` argument must be passed explicitly by name or `regex` must be a nested dictionary. The previous example, in this case, would then be:

```
In [117]: df.replace(regex=[r'\s*\.\s*', r'a|b'], value=np.nan)
Out[117]:
```

|   | a | b   | c   |
|---|---|-----|-----|
| 0 | 0 | NaN | NaN |
| 1 | 1 | NaN | NaN |
| 2 | 2 | NaN | NaN |
| 3 | 3 | NaN | d   |

This can be convenient if you do not want to pass `regex=True` every time you want to use a regular expression.

---

**Note:** Anywhere in the above `replace` examples that you see a regular expression a compiled regular expression is valid as well.

---

## 15.5.7 Numeric Replacement

`replace()` is similar to `fillna()`.

```
In [118]: df = pd.DataFrame(np.random.randn(10, 2))
In [119]: df[np.random.rand(df.shape[0]) > 0.5] = 1.5
```

(continues on next page)

Out [120]:

```
0 -0.844214 -1.021415
1  0.432396 -0.323580
2  0.423825  0.799180
3  1.262614  0.751965
4         NaN         NaN
5         NaN         NaN
6 -0.498174 -1.060799
7  0.591667 -0.183257
8  1.019855 -1.482465
9         NaN         NaN
```

Out [122]:

|   |           |           |
|---|-----------|-----------|
| 0 | a         | -1.02141  |
| 1 | 0.432396  | -0.32358  |
| 2 | 0.423825  | 0.79918   |
| 3 | 1.26261   | 0.751965  |
| 4 | NaN       | NaN       |
| 5 | NaN       | NaN       |
| 6 | -0.498174 | -1.0608   |
| 7 | 0.591667  | -0.183257 |
| 8 | 1.01985   | -1.48247  |
| 9 | NaN       | NaN       |

```
dtype('float64')
```

---

```
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

\_\_\_\_\_

|    |   |   |   |   |   |   |   |   |
|----|---|---|---|---|---|---|---|---|
| II | 1 | 1 | : | : | 1 | 1 | : | 1 |
|----|---|---|---|---|---|---|---|---|

```
In [125]: s = pd.Series([True, False, True])

In [126]: s.replace('a string', 'another string')
Out[126]:
0      True
1     False
2      True
dtype: bool
```

the original NDFrame object will be returned untouched. We’re working on unifying this API, but for backwards compatibility reasons we cannot break the latter behavior. See [GH6354](#) for more details.

## 15.6 Missing data casting rules and indexing

While pandas supports storing arrays of integer and boolean type, these types are not capable of storing missing data. Until we can switch to using a native NA type in NumPy, we’ve established some “casting rules”. When a reindexing operation introduces missing data, the Series will be cast according to the rules introduced in the table below.

| data type | Cast to |
|-----------|---------|
| integer   | float   |
| boolean   | object  |
| float     | no cast |
| object    | no cast |

For example:

```
In [127]: s = pd.Series(np.random.randn(5), index=[0, 2, 4, 6, 7])

In [128]: s > 0
Out[128]:
0      True
2      True
4      True
6      True
7      True
dtype: bool

In [129]: (s > 0).dtype
Out[129]: dtype('bool')
```

```
In [130]: crit = (s > 0).reindex(list(range(8)))

In [131]: crit
Out[131]:
0      True
1     NaN
2      True
3     NaN
4      True
5     NaN
6      True
```

(continues on next page)



(continued from previous page)

```
7      True
dtype: object
```

```
In [132]: crit.dtype
```

Ordinarily NumPy will complain if you try to use an object array (even if it contains boolean values) instead of a boolean array to get or set values from an ndarray (e.g. selecting values based on some criteria). If a boolean vector contains NAs, an exception will be generated:

```
In [133]: reindexed = s.reindex(list(range(8))).fillna(0)
```

```
In [134]: reindexed[crit]
```

[illegible]

```
<ipython-input-134-0dac417a4890> in <module>()
```

```
----> 1 reindexed[crit]
```

```
~/sandbox/pandas-release/pandas-docs/pandas/core/series.py in __getitem__(self, key)
```

```
805         key = list(key)
```

806

```
--> 807         if com.is_bool_indexer(key):
```

```
808         key = check_bool_indexer(self.index, key)
```

809

```
~/sandbox/pandas-release/pandas-docs/pandas/core/common.py in is_bool_indexer(key)
```

```
105         if not lib.is_bool_array(key):
```

```
106         if isna(key).any():
```

```
--> 107         raise ValueError('cannot index with vector containing '
```

```
108                                     'NA / NaN values')
```

```
109         return False
```

```
ValueError: cannot index with vector containing NA / NaN values
```

However, these can be filled in using `fillna()` and it will work fine:

```
In [135]: reindexed[crit.fillna(False)]
```

Out [135] :

|   |          |
|---|----------|
| 0 | 0.126504 |
|---|----------|

|   |          |
|---|----------|
| 2 | 0.696198 |
|---|----------|

|   |          |
|---|----------|
| 4 | 0.697416 |
|---|----------|

|   |          |
|---|----------|
| 6 | 0.601516 |
|---|----------|

|   |          |
|---|----------|
| 7 | 0.003659 |
|---|----------|

```
dtype: float64
```

```
In [136]: reindexed[crit.fillna(True)]
```

→

|   |          |
|---|----------|
| 0 | 0.126504 |
|---|----------|

|   |          |
|---|----------|
| 1 | 0.000000 |
|---|----------|

|   |          |
|---|----------|
| 2 | 0.696198 |
|---|----------|

|   |           |
|---|-----------|
| 3 | 0.0000000 |
|---|-----------|

|   |          |
|---|----------|
| 4 | 0.697416 |
|---|----------|

|   |          |
|---|----------|
| 5 | 0.000000 |
|---|----------|

|   |          |
|---|----------|
| 6 | 0.601516 |
|---|----------|

(continues on next page)

(continued from previous page)

```
7      0.003659  
dtype: float64
```

## GROUP BY: SPLIT-APPLY-COMBINE

By “group by” we are referring to a process involving one or more of the following steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a data structure.

Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups. In the apply step, we might wish to one of the following:

- **Aggregation:** compute a summary statistic (or statistics) for each group. Some examples:
  - Compute group sums or means.
  - Compute group sizes / counts.
- **Transformation:** perform some group-specific computations and return a like-indexed object. Some examples:
  - Standardize data (zscore) within a group.
  - Filling NAs within groups with a value derived from each group.
- **Filtration:** discard some groups, according to a group-wise computation that evaluates True or False. Some examples:
  - Discard data that belongs to groups with only a few members.
  - Filter out data based on the group sum or mean.
- Some combination of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it doesn’t fit into either of the above two categories.

Since the set of object instance methods on pandas data structures are generally rich and expressive, we often simply want to invoke, say, a DataFrame function on each group. The name GroupBy should be quite familiar to those who have used a SQL-based tool (or `itertools`), in which you can write code like:

```
SELECT Column1, Column2, mean(Column3), sum(Column4)
FROM SomeTable
GROUP BY Column1, Column2
```

We aim to make operations like this natural and easy to express using pandas. We’ll address each area of GroupBy functionality then provide some non-trivial examples / use cases.

See the [cookbook](#) for some advanced strategies.

## 16.1 Splitting an object into groups

pandas objects can be split on any of their axes. The abstract definition of grouping is to provide a mapping of labels to group names. To create a GroupBy object (more on what the GroupBy object is later), you may do the following:

```
# default is axis=0
>>> grouped = obj.groupby(key)
>>> grouped = obj.groupby(key, axis=1)
>>> grouped = obj.groupby([key1, key2])
```

The mapping can be specified many different ways:

- A Python function, to be called on each of the axis labels.
- A list or NumPy array of the same length as the selected axis.
- A dict or Series, providing a label -> group name mapping.
- For DataFrame objects, a string indicating a column to be used to group. Of course `df.groupby('A')` is just syntactic sugar for `df.groupby(df['A'])`, but it makes life simpler.
- For DataFrame objects, a string indicating an index level to be used to group.
- A list of any of the above things.

Collectively we refer to the grouping objects as the **keys**. For example, consider the following DataFrame:

---

**Note:** New in version 0.20.

A string passed to `groupby` may refer to either a column or an index level. If a string matches both a column name and an index level name then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

---

```
In [1]: df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
...:                             'foo', 'bar', 'foo', 'foo'],
...:                      'B' : ['one', 'one', 'two', 'three',
...:                             'two', 'two', 'one', 'three'],
...:                      'C' : np.random.randn(8),
...:                      'D' : np.random.randn(8)})
...:
```

```
In [2]: df
```

```
Out[2]:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 0 | foo | one   | 0.469112  | -0.861849 |
| 1 | bar | one   | -0.282863 | -2.104569 |
| 2 | foo | two   | -1.509059 | -0.494929 |
| 3 | bar | three | -1.135632 | 1.071804  |
| 4 | foo | two   | 1.212112  | 0.721555  |
| 5 | bar | two   | -0.173215 | -0.706771 |
| 6 | foo | one   | 0.119209  | -1.039575 |
| 7 | foo | three | -1.044236 | 0.271860  |

On a DataFrame, we obtain a GroupBy object by calling `groupby()`. We could naturally group by either the A or B columns, or both:

```
In [3]: grouped = df.groupby('A')
```

```
In [4]: grouped = df.groupby(['A', 'B'])
```

These will split the DataFrame on its index (rows). We could also split by the columns:

```
In [5]: def get_letter_type(letter):
...:     if letter.lower() in 'aeiou':
...:         return 'vowel'
...:     else:
...:         return 'consonant'
...:
```

```
In [6]: grouped = df.groupby(get_letter_type, axis=1)
```

pandas *Index* objects support duplicate values. If a non-unique index is used as the group key in a groupby operation, all values for the same index value will be considered to be in one group and thus the output of aggregation functions will only contain unique index values:

```
In [7]: lst = [1, 2, 3, 1, 2, 3]
```

```
In [8]: s = pd.Series([1, 2, 3, 10, 20, 30], lst)
```

```
In [9]: grouped = s.groupby(level=0)
```

```
In [10]: grouped.first()
```

Out[10]:

```
In [11]: grouped.last()
```

```
1    10
2    20
3    30
dtype: int64
```

```
In [12]: grouped.sum()
```

Note that **no splitting occurs** until it's needed. Creating the `GroupBy` object only verifies that you've passed a valid mapping.

**Note:** Many kinds of complicated data manipulations can be expressed in terms of GroupBy operations (though can't be guaranteed to be the most efficient). You can get quite creative with the label mapping functions.

### 16.1.1 GroupBy sorting

By default the group keys are sorted during the `groupby` operation. You may however pass `sort=False` for potential speedups:

```
In [13]: df2 = pd.DataFrame({'X' : ['B', 'B', 'A', 'A'], 'Y' : [1, 2, 3, 4]})
```

```
In [14]: df2.groupby(['X']).sum()
```

Out [14] :

Y

X

A 7

B 3

```
In [15]: df2.groupby(['X'], sort=False).sum()
```

```
\\Out[15]:
```

Y

X

B

A 7

Note that `groupby` will preserve the order in which *observations* are sorted *within* each group. For example, the groups created by `groupby()` below are in the order they appeared in the original `DataFrame`:

```
In [16]: df3 = pd.DataFrame({'X' : ['A', 'B', 'A', 'B'], 'Y' : [1, 4, 3, 2]})
```

```
In [17]: df3.groupby(['X']).get_group('A')
```

Out [17] :

|  | X | Y |
|--|---|---|
|--|---|---|

|   |   |   |
|---|---|---|
| 0 | A | 1 |
|---|---|---|

2 A 3

```
In [18]: df3.groupby(['X']).get_group('B')
```

```
Out[18]:
```

|  | X | Y |
|--|---|---|
|--|---|---|

1 B 4

3 B 2

### 16.1.2 GroupBy object attributes

The `groups` attribute is a dict whose keys are the computed unique groups and corresponding values being the axis labels belonging to each group. In the above example we have:

```
In [19]: df.groupby('A').groups
```

Out [19] :

```
{'bar': Int64Index([1, 3, 5], dtype='int64'),  
 'foo': Int64Index([0, 2, 4, 6, 7], dtype='int64')}
```

```
In [20]: df.groupby(get_letter_type, axis=1).groups
```

////////////////////////////////////



```
{'consonant': Index(['B', 'C', 'D'], dtype='object'),
 'vowel': Index(['A'], dtype='object')}
```

Calling the standard Python `len` function on the `GroupBy` object just returns the length of the `groups` dict, so it is largely just a convenience:

```
In [25]: qb = df.groupby('gender')
```

|              |              |            |             |             |              |
|--------------|--------------|------------|-------------|-------------|--------------|
| gb.agg       | gb.boxplot   | gb.cummin  | gb.describe | gb.filter   | gb.get_group |
| ↳ gb.height  | gb.last      | gb.median  | gb.ngroups  | gb.plot     | gb.rank      |
| ↳ gb.std     | gb.transform |            |             |             |              |
| gb.aggregate | gb.count     | gb.cumprod | gb.dtypes   | gb.first    | gb.groups    |
| ↳ gb.hist    | gb.max       | gb.min     | gb.nth      | gb.prod     | gb.resample  |
| ↳ gb.sum     | gb.var       |            |             |             |              |
| gb.apply     | gb.cummax    | gb.cumsum  | gb.fillna   | gb.gender   | gb.head      |
| ↳ gb.indices | gb.mean      | gb.name    | gb.ohlcv    | gb.quantile | gb.size      |
| ↳ gb.tail    | gb.weight    |            |             |             |              |

Let's create a Series with a two-level `MultiIndex`.

```
In [28]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])
```

841

(continued from previous page)

```
In [29]: s = pd.Series(np.random.randn(8), index=index)

In [30]: s
Out[30]:
first second
bar      one    -0.919854
          two    -0.042379
baz      one     1.247642
          two    -0.009920
foo      one     0.290213
          two     0.495767
qux      one     0.362949
          two     1.548106
dtype: float64
```

We can then group by one of the levels in `s`.

```
In [31]: grouped = s.groupby(level=0)

In [32]: grouped.sum()
Out[32]:
first
bar    -0.962232
baz     1.237723
foo     0.785980
qux     1.911055
dtype: float64
```

If the MultiIndex has names specified, these can be passed instead of the level number:

```
In [33]: s.groupby(level='second').sum()
Out[33]:
second
one     0.980950
two     1.991575
dtype: float64
```

The aggregation functions such as `sum` will take the level parameter directly. Additionally, the resulting index will be named according to the chosen level:

```
In [34]: s.sum(level='second')
Out[34]:
second
one     0.980950
two     1.991575
dtype: float64
```

Grouping with multiple levels is supported.

```
In [35]: s
Out[35]:
first second third
bar      doo    one   -1.131345
          two   -0.089329
baz      bee    one    0.337863
          two   -0.945867
```

(continues on next page)



(continued from previous page)

```

foo    bop    one    -0.932132
        two     1.956030
qux    bop    one     0.017587
        two    -0.016692
dtype: float64

```

```
In [36]: s.groupby(level=['first', 'second']).sum()
```

```

////////////////////////////////////
↪
first  second
bar    doo    -1.220674
baz    bee    -0.608004
foo    bop     1.023898
qux    bop     0.000895
dtype: float64

```

New in version 0.20.

Index level names may be supplied as keys.

```
In [37]: s.groupby(['first', 'second']).sum()
```

```

Out [37]:
first  second
bar    doo    -1.220674
baz    bee    -0.608004
foo    bop     1.023898
qux    bop     0.000895
dtype: float64

```

More on the `sum` function and aggregation later.

### 16.1.4 Grouping DataFrame with Index Levels and Columns

A DataFrame may be grouped by a combination of columns and index levels by specifying the column names as strings and the index levels as `pd.Grouper` objects.

```
In [38]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
.....:             ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]
.....:
```

```
In [39]: index = pd.MultiIndex.from_arrays(arrays, names=['first', 'second'])
```

```
In [40]: df = pd.DataFrame({'A': [1, 1, 1, 1, 2, 2, 3, 3],
.....:                     'B': np.arange(8)},
.....:                     index=index)
.....:
```

```
In [41]: df
```

```

Out [41]:
      A  B
first second
bar  one  1  0
     two  1  1
baz  one  1  2
     two  1  3

```

(continues on next page)

(continued from previous page)

|     |     |   |   |
|-----|-----|---|---|
| foo | one | 2 | 4 |
|     | two | 2 | 5 |
| qux | one | 3 | 6 |
|     | two | 3 | 7 |

The following example groups df by the second index level and the A column.

```
In [42]: df.groupby([pd.Grouper(level=1), 'A']).sum()
Out[42]:
```

|        |   |   |
|--------|---|---|
|        |   | B |
| second | A |   |
| one    | 1 | 2 |
|        | 2 | 4 |
|        | 3 | 6 |
| two    | 1 | 4 |
|        | 2 | 5 |
|        | 3 | 7 |

Index levels may also be specified by name.

```
In [43]: df.groupby([pd.Grouper(level='second'), 'A']).sum()
Out[43]:
```

|        |   |   |
|--------|---|---|
|        |   | B |
| second | A |   |
| one    | 1 | 2 |
|        | 2 | 4 |
|        | 3 | 6 |
| two    | 1 | 4 |
|        | 2 | 5 |
|        | 3 | 7 |

New in version 0.20.

Index level names may be specified as keys directly to groupby.

```
In [44]: df.groupby(['second', 'A']).sum()
Out[44]:
```

|        |   |   |
|--------|---|---|
|        |   | B |
| second | A |   |
| one    | 1 | 2 |
|        | 2 | 4 |
|        | 3 | 6 |
| two    | 1 | 4 |
|        | 2 | 5 |
|        | 3 | 7 |

### 16.1.5 DataFrame column selection in GroupBy

Once you have created the GroupBy object from a DataFrame, you might want to do something different for each of the columns. Thus, using `[]` similar to getting a column from a DataFrame, you can do:

```
In [45]: grouped = df.groupby(['A'])
In [46]: grouped_C = grouped['C']
In [47]: grouped_D = grouped['D']
```

This is mainly syntactic sugar for the alternative and much more verbose:

```
In [48]: df['C'].groupby(df['A'])
Out[48]: <pandas.core.groupby.groupby.SeriesGroupBy object at 0x1c2adc8390>
```

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

## 16.2 Iterating through groups

With the GroupBy object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby()`:

```
In [49]: grouped = df.groupby('A')

In [50]: for name, group in grouped:
.....:     print(name)
.....:     print(group)
.....:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 1 | bar | one   | 0.254161  | 1.511763  |
| 3 | bar | three | 0.215897  | -0.990582 |
| 5 | bar | two   | -0.077118 | 1.211526  |

foo

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 0 | foo | one   | -0.575247 | 1.346061  |
| 2 | foo | two   | -1.143704 | 1.627081  |
| 4 | foo | two   | 1.193555  | -0.441652 |
| 6 | foo | one   | -0.408530 | 0.268520  |
| 7 | foo | three | -0.862495 | 0.024580  |

In the case of grouping by multiple keys, the group name will be a tuple:

```
In [51]: for name, group in df.groupby(['A', 'B']):
.....:     print(name)
.....:     print(group)
.....:
```

('bar', 'one')

|   | A   | B   | C        | D        |
|---|-----|-----|----------|----------|
| 1 | bar | one | 0.254161 | 1.511763 |

('bar', 'three')

|   | A   | B     | C        | D         |
|---|-----|-------|----------|-----------|
| 3 | bar | three | 0.215897 | -0.990582 |

('bar', 'two')

|   | A   | B   | C         | D        |
|---|-----|-----|-----------|----------|
| 5 | bar | two | -0.077118 | 1.211526 |

('foo', 'one')

|   | A   | B   | C         | D        |
|---|-----|-----|-----------|----------|
| 0 | foo | one | -0.575247 | 1.346061 |
| 6 | foo | one | -0.408530 | 0.268520 |

('foo', 'three')

|   | A   | B     | C         | D       |
|---|-----|-------|-----------|---------|
| 7 | foo | three | -0.862495 | 0.02458 |

('foo', 'two')

|  | A | B | C | D |
|--|---|---|---|---|
|--|---|---|---|---|

(continues on next page)

(continued from previous page)

```
2  foo  two -1.143704  1.627081
4  foo  two  1.193555 -0.441652
```

It's standard Python-fu but remember you can unpack the tuple in the for loop statement if you wish: `for (k1, k2), group in grouped:`.

## 16.3 Selecting a group

A single group can be selected using `get_group()`:

```
In [52]: grouped.get_group('bar')
Out[52]:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 1 | bar | one   | 0.254161  | 1.511763  |
| 3 | bar | three | 0.215897  | -0.990582 |
| 5 | bar | two   | -0.077118 | 1.211526  |

Or for an object grouped on multiple columns:

```
In [53]: df.groupby(['A', 'B']).get_group(('bar', 'one'))
Out[53]:
```

|   | A   | B   | C        | D        |
|---|-----|-----|----------|----------|
| 1 | bar | one | 0.254161 | 1.511763 |

## 16.4 Aggregation

Once the `GroupBy` object has been created, several methods are available to perform a computation on the grouped data. These operations are similar to the *aggregating API*, *window functions API*, and *resample API*.

An obvious one is aggregation via the `aggregate()` or equivalently `agg()` method:

```
In [54]: grouped = df.groupby('A')

In [55]: grouped.agg(np.sum)
Out[55]:
```

|     |  | C         | D        |
|-----|--|-----------|----------|
| A   |  |           |          |
| bar |  | 0.392940  | 1.732707 |
| foo |  | -1.796421 | 2.824590 |

```
In [56]: grouped = df.groupby(['A', 'B'])

In [57]: grouped.agg(np.sum)
Out[57]:
```

|     |       |  | C         | D         |
|-----|-------|--|-----------|-----------|
| A   | B     |  |           |           |
| bar | one   |  | 0.254161  | 1.511763  |
|     | three |  | 0.215897  | -0.990582 |
|     | two   |  | -0.077118 | 1.211526  |
| foo | one   |  | -0.983776 | 1.614581  |
|     | three |  | -0.862495 | 0.024580  |
|     | two   |  | 0.049851  | 1.185429  |

As you can see, the result of the aggregation will have the group names as the new index along the grouped axis. In the case of multiple keys, the result is a *MultiIndex* by default, though this can be changed by using the `as_index` option:

```
In [58]: grouped = df.groupby(['A', 'B'], as_index=False)
```

```
In [59]: grouped.agg(np.sum)
```

```
Out[59]:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 0 | bar | one   | 0.254161  | 1.511763  |
| 1 | bar | three | 0.215897  | -0.990582 |
| 2 | bar | two   | -0.077118 | 1.211526  |
| 3 | foo | one   | -0.983776 | 1.614581  |
| 4 | foo | three | -0.862495 | 0.024580  |
| 5 | foo | two   | 0.049851  | 1.185429  |

```
In [60]: df.groupby('A', as_index=False).sum()
```

```

////////////////////////////////////
↪
      A      C      D
0  bar  0.392940  1.732707
1  foo -1.796421  2.824590

```

Note that you could use the `reset_index` DataFrame function to achieve the same result as the column names are stored in the resulting *MultiIndex*:

```
In [61]: df.groupby(['A', 'B']).sum().reset_index()
```

```
Out[61]:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 0 | bar | one   | 0.254161  | 1.511763  |
| 1 | bar | three | 0.215897  | -0.990582 |
| 2 | bar | two   | -0.077118 | 1.211526  |
| 3 | foo | one   | -0.983776 | 1.614581  |
| 4 | foo | three | -0.862495 | 0.024580  |
| 5 | foo | two   | 0.049851  | 1.185429  |

Another simple aggregation example is to compute the size of each group. This is included in *GroupBy* as the `size` method. It returns a *Series* whose index are the group names and whose values are the sizes of each group.

```
In [62]: grouped.size()
```

```
Out[62]:
```

```

A      B
bar  one      1
      three   1
      two     1
foo   one     2
      three   1
      two     2
dtype: int64

```

```
In [63]: grouped.describe()
```

```
Out[63]:
```

|   | C      |          |          |          |          | ... | D        |          |     |  |  |
|---|--------|----------|----------|----------|----------|-----|----------|----------|-----|--|--|
|   | count  | mean     | std      | min      | 25%      | ... | min      | 25%      | 50% |  |  |
| ↪ | count  | mean     | std      | min      | 25%      | ... | min      | 25%      | 50% |  |  |
| ↪ | %      | 75%      | max      |          |          |     |          |          |     |  |  |
| 0 | 1.0    | 0.254161 | NaN      | 0.254161 | 0.254161 | ... | 1.511763 | 1.511763 | 1.  |  |  |
| ↪ | 511763 | 1.511763 | 1.511763 |          |          |     |          |          |     |  |  |

(continues on next page)

(continued from previous page)

```

1    1.0    0.215897      NaN    0.215897    0.215897    ...    -0.990582 -0.990582 -0.
↪990582 -0.990582 -0.990582
2    1.0   -0.077118      NaN   -0.077118   -0.077118    ...      1.211526  1.211526  1.
↪211526  1.211526  1.211526
3    2.0   -0.491888    0.117887  -0.575247  -0.533567    ...      0.268520  0.537905  0.
↪807291  1.076676  1.346061
4    1.0   -0.862495      NaN   -0.862495   -0.862495    ...      0.024580  0.024580  0.
↪024580  0.024580  0.024580
5    2.0    0.024925    1.652692  -1.143704  -0.559389    ...     -0.441652  0.075531  0.
↪592714  1.109898  1.627081

[6 rows x 16 columns]
```

**Note:** Aggregation functions **will not** return the groups that you are aggregating over if they are named *columns*, when `as_index=True`, the default. The grouped columns will be the **indices** of the returned object.

Passing `as_index=False` **will** return the groups that you are aggregating over, if they are named *columns*.

Aggregating functions are the ones that reduce the dimension of the returned objects. Some common aggregating functions are tabulated below:

| Function                | Description  |
|-------------------------|--|
| <code>mean()</code>     | Compute mean of groups                                   |
| <code>sum()</code>      | Compute sum of group values                              |
| <code>size()</code>     | Compute group sizes                                      |
| <code>count()</code>    | Compute count of group                                   |
| <code>std()</code>      | Standard deviation of groups                             |
| <code>var()</code>      | Compute variance of groups                               |
| <code>sem()</code>      | Standard error of the mean of groups                     |
| <code>describe()</code> | Generates descriptive statistics                         |
| <code>first()</code>    | Compute first of group values                            |
| <code>last()</code>     | Compute last of group values                             |
| <code>nth()</code>      | Take <i>nth</i> value, or a subset if <i>n</i> is a list |
| <code>min()</code>      | Compute min of group values                              |
| <code>max()</code>      | Compute max of group values                              |

The aggregating functions above will exclude NA values. Any function which reduces a *Series* to a scalar value is an aggregation function and will work, a trivial example is `df.groupby('A').agg(lambda ser: 1)`. Note that `nth()` can act as a reducer *or* a filter, see [here](#).

### 16.4.1 Applying multiple functions at once

With grouped *Series* you can also pass a list or dict of functions to do aggregation with, outputting a *DataFrame*:

```

In [64]: grouped = df.groupby('A')

In [65]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[65]:
```

|   | sum | mean | std |
|---|-----|------|-----|
| A |     |      |     |

(continues on next page)

(continued from previous page)

```
bar  0.392940  0.130980  0.181231
foo -1.796421 -0.359284  0.912265
```

On a grouped DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

```
In [66]: grouped.agg([np.sum, np.mean, np.std])
Out[66]:
```

|     | C         |           |          | D        |          |          |
|-----|-----------|-----------|----------|----------|----------|----------|
|     | sum       | mean      | std      | sum      | mean     | std      |
| A   |           |           |          |          |          |          |
| bar | 0.392940  | 0.130980  | 0.181231 | 1.732707 | 0.577569 | 1.366330 |
| foo | -1.796421 | -0.359284 | 0.912265 | 2.824590 | 0.564918 | 0.884785 |

The resulting aggregations are named for the functions themselves. If you need to rename, then you can add in a chained operation for a Series like this:

```
In [67]: (grouped['C'].agg([np.sum, np.mean, np.std])
.....:                .rename(columns={'sum': 'foo',
.....:                                'mean': 'bar',
.....:                                'std': 'baz'}))
.....: )
Out[67]:
```

|     | foo       | bar       | baz      |
|-----|-----------|-----------|----------|
| A   |           |           |          |
| bar | 0.392940  | 0.130980  | 0.181231 |
| foo | -1.796421 | -0.359284 | 0.912265 |

For a grouped DataFrame, you can rename in a similar manner:

```
In [68]: (grouped.agg([np.sum, np.mean, np.std])
.....:                .rename(columns={'sum': 'foo',
.....:                                'mean': 'bar',
.....:                                'std': 'baz'}))
.....: )
Out[68]:
```

|     | C         |           |          | D        |          |          |
|-----|-----------|-----------|----------|----------|----------|----------|
|     | foo       | bar       | baz      | foo      | bar      | baz      |
| A   |           |           |          |          |          |          |
| bar | 0.392940  | 0.130980  | 0.181231 | 1.732707 | 0.577569 | 1.366330 |
| foo | -1.796421 | -0.359284 | 0.912265 | 2.824590 | 0.564918 | 0.884785 |

## 16.4.2 Applying different functions to DataFrame columns

By passing a dict to aggregate you can apply a different aggregation to the columns of a DataFrame:

```
In [69]: grouped.agg({'C' : np.sum,
.....:                'D' : lambda x: np.std(x, ddof=1)})
.....:
Out[69]:
```

|   | C | D |
|---|---|---|
| A |   |   |

(continues on next page)

(continued from previous page)

```
bar 0.392940 1.366330
foo -1.796421 0.884785
```

The function names can also be strings. In order for a string to be valid it must be either implemented on GroupBy or available via *dispatching*:

```
In [70]: grouped.agg({'C' : 'sum', 'D' : 'std'})
Out[70]:
```

|     | C         | D        |
|-----|-----------|----------|
| A   |           |          |
| bar | 0.392940  | 1.366330 |
| foo | -1.796421 | 0.884785 |

**Note:** If you pass a dict to `aggregate`, the ordering of the output columns is non-deterministic. If you want to be sure the output columns will be in a specific order, you can use an `OrderedDict`. Compare the output of the following two commands:

```
In [71]: grouped.agg({'D': 'std', 'C': 'mean'})
Out[71]:
```

|     | D        | C         |
|-----|----------|-----------|
| A   |          |           |
| bar | 1.366330 | 0.130980  |
| foo | 0.884785 | -0.359284 |

```
In [72]: grouped.agg(OrderedDict([('D', 'std'), ('C', 'mean')]))
////////////////////////////////////
```

|     | D        | C         |
|-----|----------|-----------|
| A   |          |           |
| bar | 1.366330 | 0.130980  |
| foo | 0.884785 | -0.359284 |

### 16.4.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, `std`, and `sem`, have optimized Cython implementations:

```
In [73]: df.groupby('A').sum()
Out[73]:
```

|     | C         | D        |
|-----|-----------|----------|
| A   |           |          |
| bar | 0.392940  | 1.732707 |
| foo | -1.796421 | 2.824590 |

```
In [74]: df.groupby(['A', 'B']).mean()
///////////////////////////////////////////////////
↪
```

|     |       | C         | D         |
|-----|-------|-----------|-----------|
| A   | B     |           |           |
| bar | one   | 0.254161  | 1.511763  |
|     | three | 0.215897  | -0.990582 |
|     | two   | -0.077118 | 1.211526  |
| foo | one   | -0.491888 | 0.807291  |

(continues on next page)



|       |           |          |
|-------|-----------|----------|
| three | -0.862495 | 0.024580 |
| two   | 0.024925  | 0.592714 |

```
In [75]: index = pd.date_range('10/1/1999', periods=1100)
In [76]: ts = pd.Series(np.random.normal(0.5, 2, 1100), index)
In [77]: ts = ts.rolling(window=100,min_periods=100).mean().dropna()
```

```
2000-01-08    0.779333
2000-01-09    0.778852
2000-01-10    0.786476
2000-01-11    0.782797
2000-01-12    0.798110
Freq: D, dtype: float64
```

```

2002-09-30    0.660294
2002-10-01    0.631095
2002-10-02    0.673601
2002-10-03    0.709213
2002-10-04    0.719369
Freq: D, dtype: float64

```

We would expect the result to now have mean 0 and standard deviation 1 within each group, which we can easily check:

```
# Original Data
In [83]: grouped = ts.groupby(key)

In [84]: grouped.mean()
Out[84]:
2000    0.442441
2001    0.526246
2002    0.459365
dtype: float64

In [85]: grouped.std()
Out[85]:
2000    0.131752
2001    0.210945
2002    0.128753
dtype: float64

# Transformed Data
In [86]: grouped_trans = transformed.groupby(key)

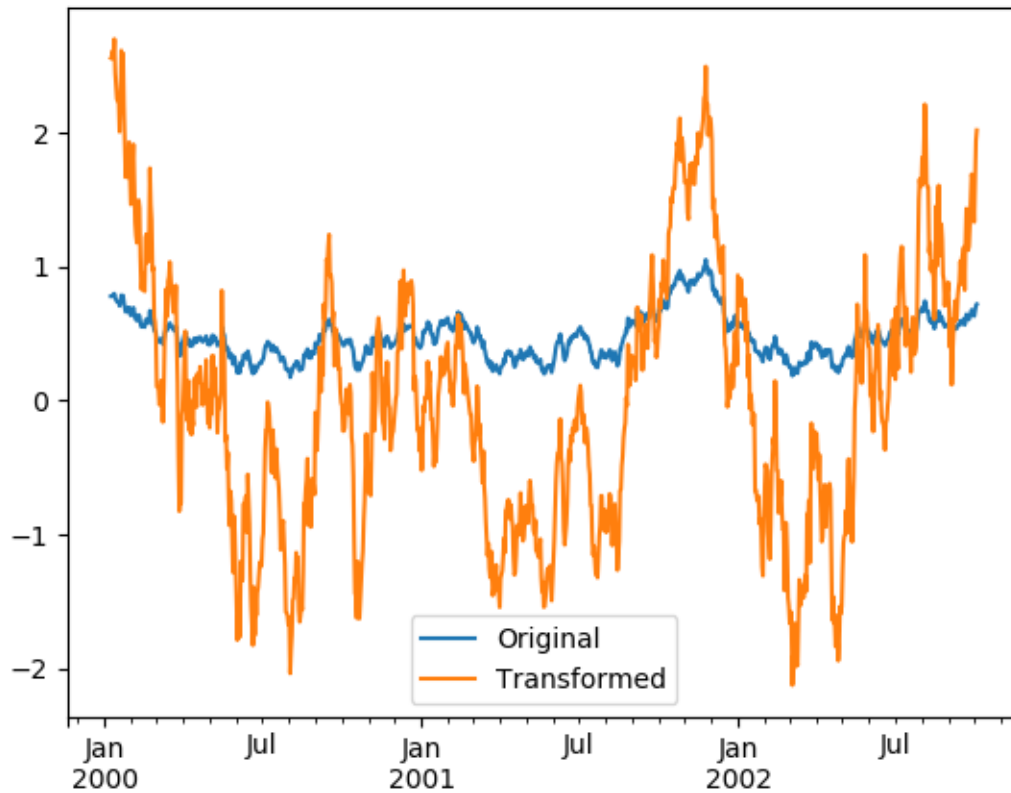
In [87]: grouped_trans.mean()
Out[87]:
2000    1.168208e-15
2001    1.454544e-15
2002    1.726657e-15
dtype: float64

In [88]: grouped_trans.std()
Out[88]:
2000    1.0
2001    1.0
2002    1.0
dtype: float64
```

We can also visually compare the original and transformed data sets.

```
In [89]: compare = pd.DataFrame({'Original': ts, 'Transformed': transformed})

In [90]: compare.plot()
Out[90]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2ad9db38>
```



Transformation functions that have lower dimension outputs are broadcast to match the shape of the input array.

```
In [91]: data_range = lambda x: x.max() - x.min()
```

```
In [92]: ts.groupby(key).transform(data_range)
```

```
Out[92]:
```

```
2000-01-08    0.623893
2000-01-09    0.623893
2000-01-10    0.623893
2000-01-11    0.623893
2000-01-12    0.623893
2000-01-13    0.623893
2000-01-14    0.623893
...
2002-09-28    0.558275
2002-09-29    0.558275
2002-09-30    0.558275
2002-10-01    0.558275
2002-10-02    0.558275
2002-10-03    0.558275
2002-10-04    0.558275
Freq: D, Length: 1001, dtype: float64
```

Alternatively the built-in methods can be used to produce the same outputs

```
In [93]: ts.groupby(key).transform('max') - ts.groupby(key).transform('min')
Out[93]:
2000-01-08    0.623893
2000-01-09    0.623893
2000-01-10    0.623893
2000-01-11    0.623893
2000-01-12    0.623893
2000-01-13    0.623893
2000-01-14    0.623893
...
2002-09-28    0.558275
2002-09-29    0.558275
2002-09-30    0.558275
2002-10-01    0.558275
2002-10-02    0.558275
2002-10-03    0.558275
2002-10-04    0.558275
Freq: D, Length: 1001, dtype: float64
```

Another common data transform is to replace missing data with the group mean.

```
In [94]: data_df
Out[94]:
      A      B      C
0  1.539708 -1.166480  0.533026
1  1.302092 -0.505754    NaN
2 -0.371983  1.104803 -0.651520
3 -1.309622  1.118697 -1.161657
4 -1.924296  0.396437  0.812436
5  0.815643  0.367816 -0.469478
6 -0.030651  1.376106 -0.645129
..    ...    ...    ...
993  0.012359  0.554602 -1.976159
994  0.042312 -1.628835  1.013822
995 -0.093110  0.683847 -0.774753
996 -0.185043  1.438572    NaN
997 -0.394469 -0.642343  0.011374
998 -1.174126  1.857148    NaN
999  0.234564  0.517098  0.393534

[1000 rows x 3 columns]

In [95]: countries = np.array(['US', 'UK', 'GR', 'JP'])

In [96]: key = countries[np.random.randint(0, 4, 1000)]

In [97]: grouped = data_df.groupby(key)

# Non-NA count in each group
In [98]: grouped.count()
Out[98]:
      A      B      C
GR  209  217  189
JP  240  255  217
UK  216  231  193
US  239  250  217
```

(continues on next page)



For example: `fillna`, `ffill`, `bfill`, `shift`..

```
In [107]: grouped.ffmpeg()
Out[107]:
```

|     | NaN | A         | B         | C         |
|-----|-----|-----------|-----------|-----------|
| 0   | US  | 1.539708  | -1.166480 | 0.533026  |
| 1   | US  | 1.302092  | -0.505754 | 0.533026  |
| 2   | US  | -0.371983 | 1.104803  | -0.651520 |
| 3   | JP  | -1.309622 | 1.118697  | -1.161657 |
| 4   | JP  | -1.924296 | 0.396437  | 0.812436  |
| 5   | US  | 0.815643  | 0.367816  | -0.469478 |
| 6   | GR  | -0.030651 | 1.376106  | -0.645129 |
| ..  | ..  | ...       | ...       | ...       |
| 993 | US  | 0.012359  | 0.554602  | -1.976159 |
| 994 | GR  | 0.042312  | -1.628835 | 1.013822  |
| 995 | JP  | -0.093110 | 0.683847  | -0.774753 |
| 996 | JP  | -0.185043 | 1.438572  | -0.774753 |
| 997 | GR  | -0.394469 | -0.642343 | 0.011374  |
| 998 | JP  | -1.174126 | 1.857148  | -0.774753 |
| 999 | UK  | 0.234564  | 0.517098  | 0.393534  |

[1000 rows x 4 columns]

## 16.5.1 New syntax to window and resample operations

New in version 0.18.1.

Working with the `resample`, `expanding` or `rolling` operations on the `groupby` level used to require the application of helper functions. However, now it is possible to use `resample()`, `expanding()` and `rolling()` as methods on `groupbys`.

The example below will apply the `rolling()` method on the samples of the column `B` based on the groups of column `A`.

```
In [108]: df_re = pd.DataFrame({'A': [1] * 10 + [5] * 10,
.....:                        'B': np.arange(20)})
.....:

In [109]: df_re
Out[109]:
```

|    | A  | B  |
|----|----|----|
| 0  | 1  | 0  |
| 1  | 1  | 1  |
| 2  | 1  | 2  |
| 3  | 1  | 3  |
| 4  | 1  | 4  |
| 5  | 1  | 5  |
| 6  | 1  | 6  |
| .. | .. | .. |
| 13 | 5  | 13 |
| 14 | 5  | 14 |
| 15 | 5  | 15 |
| 16 | 5  | 16 |
| 17 | 5  | 17 |
| 18 | 5  | 18 |
| 19 | 5  | 19 |

(continues on next page)

(continued from previous page)

[20 rows x 2 columns]

```
In [110]: df_re.groupby('A').rolling(4).B.mean()
```

```

////////////////////////////////////

```

```

↪

```

```

A
1  0      NaN
   1      NaN
   2      NaN
   3      1.5
   4      2.5
   5      3.5
   6      4.5
   ...
5 13     11.5
   14     12.5
   15     13.5
   16     14.5
   17     15.5
   18     16.5
   19     17.5

```

Name: B, Length: 20, dtype: float64

The `expanding()` method will accumulate a given operation (`sum()` in the example) for all the members of each particular group.

```
In [111]: df_re.groupby('A').expanding().sum()
```

```
Out [111]:
```

```

      A      B
A
1  0    1.0    0.0
   1    2.0    1.0
   2    3.0    3.0
   3    4.0    6.0
   4    5.0   10.0
   5    6.0   15.0
   6    7.0   21.0
   ...
5 13   20.0   46.0
   14   25.0   60.0
   15   30.0   75.0
   16   35.0   91.0
   17   40.0  108.0
   18   45.0  126.0
   19   50.0  145.0

```

[20 rows x 2 columns]

Suppose you want to use the `resample()` method to get a daily frequency in each group of your dataframe and wish to complete the missing values with the `ffill()` method.

```

In [112]: df_re = pd.DataFrame({'date': pd.date_range(start='2016-01-01',
.....:                                                  periods=4,
.....:                                                  freq='W'),
.....:                          'group': [1, 1, 2, 2],

```

(continues on next page)

(continued from previous page)

```
.....:                                     'val': [5, 6, 7, 8])).set_index('date')
.....:
```

```
In [113]: df_re
```

```
Out[113]:
```

|            | group | val |
|------------|-------|-----|
| date       |       |     |
| 2016-01-03 | 1     | 5   |
| 2016-01-10 | 1     | 6   |
| 2016-01-17 | 2     | 7   |
| 2016-01-24 | 2     | 8   |

```
In [114]: df_re.groupby('group').resample('1D').ffill()
```

```
////////////////////////////////////
```

|       |            | group | val |
|-------|------------|-------|-----|
| group | date       |       |     |
| 1     | 2016-01-03 | 1     | 5   |
|       | 2016-01-04 | 1     | 5   |
|       | 2016-01-05 | 1     | 5   |
|       | 2016-01-06 | 1     | 5   |
|       | 2016-01-07 | 1     | 5   |
|       | 2016-01-08 | 1     | 5   |
|       | 2016-01-09 | 1     | 5   |
| ...   |            | ...   | ... |
| 2     | 2016-01-18 | 2     | 7   |
|       | 2016-01-19 | 2     | 7   |
|       | 2016-01-20 | 2     | 7   |
|       | 2016-01-21 | 2     | 7   |
|       | 2016-01-22 | 2     | 7   |
|       | 2016-01-23 | 2     | 7   |
|       | 2016-01-24 | 2     | 8   |

```
[16 rows x 2 columns]
```

## 16.6 Filtration

The `filter` method returns a subset of the original object. Suppose we want to take only elements that belong to groups with a group sum greater than 2.

```
In [115]: sf = pd.Series([1, 1, 2, 3, 3, 3])
```

```
In [116]: sf.groupby(sf).filter(lambda x: x.sum() > 2)
```

```
Out[116]:
```

|   |   |
|---|---|
| 3 | 3 |
| 4 | 3 |
| 5 | 3 |

dtype: int64

The argument of `filter` must be a function that, applied to the group as a whole, returns `True` or `False`.

Another useful operation is filtering out elements that belong to groups with only a couple members.



```
In [117]: dff = pd.DataFrame({'A': np.arange(8), 'B': list('aabbbbcc')})

In [118]: dff.groupby('B').filter(lambda x: len(x) > 2)
Out[118]:
```

|   | A | B |
|---|---|---|
| 2 | 2 | b |
| 3 | 3 | b |
| 4 | 4 | b |
| 5 | 5 | b |

Alternatively, instead of dropping the offending groups, we can return a like-indexed objects where the groups that do not pass the filter are filled with NaNs.

```
In [119]: dff.groupby('B').filter(lambda x: len(x) > 2, dropna=False)
Out[119]:
```

|   | A   | B   |
|---|-----|-----|
| 0 | NaN | NaN |
| 1 | NaN | NaN |
| 2 | 2.0 | b   |
| 3 | 3.0 | b   |
| 4 | 4.0 | b   |
| 5 | 5.0 | b   |
| 6 | NaN | NaN |
| 7 | NaN | NaN |

For DataFrames with multiple columns, filters should explicitly specify a column as the filter criterion.

```
In [120]: dff['C'] = np.arange(8)

In [121]: dff.groupby('B').filter(lambda x: len(x['C']) > 2)
Out[121]:
```

|   | A | B | C |
|---|---|---|---|
| 2 | 2 | b | 2 |
| 3 | 3 | b | 3 |
| 4 | 4 | b | 4 |
| 5 | 5 | b | 5 |

**Note:** Some functions when applied to a groupby object will act as a **filter** on the input, returning a reduced shape of the original (and potentially eliminating groups), but with the index unchanged. Passing `as_index=False` will not affect these transformation methods.

For example: `head`, `tail`.

```
In [122]: dff.groupby('B').head(2)
Out[122]:
```

|   | A | B | C |
|---|---|---|---|
| 0 | 0 | a | 0 |
| 1 | 1 | a | 1 |
| 2 | 2 | b | 2 |
| 3 | 3 | b | 3 |
| 6 | 6 | c | 6 |
| 7 | 7 | c | 7 |

## 16.7 Dispatching to instance methods

When doing an aggregation or transformation, you might just want to call an instance method on each data group. This is pretty easy to do by passing lambda functions:

```
In [123]: grouped = df.groupby('A')

In [124]: grouped.agg(lambda x: x.std())
Out[124]:
```

|     | C        | D        |
|-----|----------|----------|
| A   |          |          |
| bar | 0.181231 | 1.366330 |
| foo | 0.912265 | 0.884785 |

But, it's rather verbose and can be untidy if you need to pass additional arguments. Using a bit of metaprogramming cleverness, GroupBy now has the ability to “dispatch” method calls to the groups:

```
In [125]: grouped.std()
Out[125]:
```

|     | C        | D        |
|-----|----------|----------|
| A   |          |          |
| bar | 0.181231 | 1.366330 |
| foo | 0.912265 | 0.884785 |

What is actually happening here is that a function wrapper is being generated. When invoked, it takes any passed arguments and invokes the function with any arguments on each group (in the above example, the `std` function). The results are then combined together much in the style of `agg` and `transform` (it actually uses `apply` to infer the gluing, documented next). This enables some operations to be carried out rather succinctly:

```
In [126]: tsdf = pd.DataFrame(np.random.randn(1000, 3),
.....:                        index=pd.date_range('1/1/2000', periods=1000),
.....:                        columns=['A', 'B', 'C'])
.....:

In [127]: tsdf.iloc[:,2] = np.nan

In [128]: grouped = tsdf.groupby(lambda x: x.year)

In [129]: grouped.fillna(method='pad')
Out[129]:
```

|            | A         | B         | C         |
|------------|-----------|-----------|-----------|
| 2000-01-01 | NaN       | NaN       | NaN       |
| 2000-01-02 | -0.353501 | -0.080957 | -0.876864 |
| 2000-01-03 | -0.353501 | -0.080957 | -0.876864 |
| 2000-01-04 | 0.050976  | 0.044273  | -0.559849 |
| 2000-01-05 | 0.050976  | 0.044273  | -0.559849 |
| 2000-01-06 | 0.030091  | 0.186460  | -0.680149 |
| 2000-01-07 | 0.030091  | 0.186460  | -0.680149 |
| ...        | ...       | ...       | ...       |
| 2002-09-20 | 2.310215  | 0.157482  | -0.064476 |
| 2002-09-21 | 2.310215  | 0.157482  | -0.064476 |
| 2002-09-22 | 0.005011  | 0.053897  | -1.026922 |
| 2002-09-23 | 0.005011  | 0.053897  | -1.026922 |
| 2002-09-24 | -0.456542 | -1.849051 | 1.559856  |
| 2002-09-25 | -0.456542 | -1.849051 | 1.559856  |
| 2002-09-26 | 1.123162  | 0.354660  | 1.128135  |

(continues on next page)

(continued from previous page)

```
[1000 rows x 3 columns]
```

In this example, we chopped the collection of time series into yearly chunks then independently called *fillna* on the groups.

The `nlargest` and `nsmallest` methods work on `Series` style groupbys:

```
In [130]: s = pd.Series([9, 8, 7, 5, 19, 1, 4.2, 3.3])
```

```
In [131]: g = pd.Series(list('abababab'))
```

```
In [132]: gb = s.groupby(g)
```

```
In [133]: gb.nlargest(3)
```

```
Out[133]:
a  4    19.0
   0     9.0
   2     7.0
b  1     8.0
   3     5.0
   7     3.3
dtype: float64
```

```
In [134]: gb.nsmallest(3)
```

```
Out[134]:
a  6     4.2
   2     7.0
   0     9.0
b  5     1.0
   7     3.3
   3     5.0
dtype: float64
```

## 16.8 Flexible apply

Some operations on the grouped data might not fit into either the aggregate or transform categories. Or, you may simply want `GroupBy` to infer how to combine the results. For these, use the `apply` function, which can be substituted for both `aggregate` and `transform` in many standard use cases. However, `apply` can handle some exceptional use cases, for example:

```
In [135]: df
```

```
Out[135]:
   A      B      C      D
0  foo  one -0.575247  1.346061
1  bar  one  0.254161  1.511763
2  foo  two -1.143704  1.627081
3  bar three  0.215897 -0.990582
4  foo  two  1.193555 -0.441652
5  bar  two -0.077118  1.211526
6  foo  one -0.408530  0.268520
7  foo three -0.862495  0.024580
```

(continues on next page)

(continued from previous page)

```
In [136]: grouped = df.groupby('A')

# could also just call .describe()
In [137]: grouped['C'].apply(lambda x: x.describe())
Out[137]:
A
bar  count      3.000000
     mean      0.130980
     std       0.181231
     min      -0.077118
     25%       0.069390
     50%       0.215897
     75%       0.235029
     ...
foo  mean      -0.359284
     std       0.912265
     min      -1.143704
     25%      -0.862495
     50%      -0.575247
     75%      -0.408530
     max       1.193555
Name: C, Length: 16, dtype: float64
```

The dimension of the returned result can also change:

```
In [138]: grouped = df.groupby('A')['C']

In [139]: def f(group):
.....:     return pd.DataFrame({'original' : group,
.....:                          'demeaned' : group - group.mean()})
.....:

In [140]: grouped.apply(f)
Out[140]:
   original  demeaned
0 -0.575247 -0.215962
1  0.254161  0.123181
2 -1.143704 -0.784420
3  0.215897  0.084917
4  1.193555  1.552839
5 -0.077118 -0.208098
6 -0.408530 -0.049245
7 -0.862495 -0.503211
```

apply on a Series can operate on a returned value from the applied function, that is itself a series, and possibly upcast the result to a DataFrame:

```
In [141]: def f(x):
.....:     return pd.Series([ x, x**2 ], index = ['x', 'x^2'])
.....:

In [142]: s
Out[142]:
0    9.0
1    8.0
2    7.0
```

(continues on next page)

(continued from previous page)

```

3      5.0
4     19.0
5      1.0
6      4.2
7      3.3
dtype: float64

```

```
In [143]: s.apply(f)
```

```

////////////////////////////////////
↪
      x      x^2
0   9.0   81.00
1   8.0   64.00
2   7.0   49.00
3   5.0   25.00
4  19.0  361.00
5   1.0    1.00
6   4.2   17.64
7   3.3   10.89

```

**Note:** `apply` can act as a reducer, transformer, *or* filter function, depending on exactly what is passed to it. So depending on the path taken, and exactly what you are grouping. Thus the grouped column(s) may be included in the output as well as set the indices.

**Warning:** In the current implementation `apply` calls `func` twice on the first group to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if `func` has side-effects, as they will take effect twice for the first group.

```
In [144]: d = pd.DataFrame({"a": ["x", "y"], "b": [1, 2]})
```

```
In [145]: def identity(df):
.....:     print(df)
.....:     return df
.....:
```

```
In [146]: d.groupby("a").apply(identity)
```

```

a  b
0  x  1
a  b
0  x  1
a  b
1  y  2

```

```
Out[146]:
```

```

a  b
0  x  1
1  y  2

```

## 16.9 Other useful features

### 16.9.1 Automatic exclusion of “nuisance” columns

Again consider the example DataFrame we’ve been looking at:

```
In [147]: df
Out[147]:
```

|   | A   | B     | C         | D         |
|---|-----|-------|-----------|-----------|
| 0 | foo | one   | -0.575247 | 1.346061  |
| 1 | bar | one   | 0.254161  | 1.511763  |
| 2 | foo | two   | -1.143704 | 1.627081  |
| 3 | bar | three | 0.215897  | -0.990582 |
| 4 | foo | two   | 1.193555  | -0.441652 |
| 5 | bar | two   | -0.077118 | 1.211526  |
| 6 | foo | one   | -0.408530 | 0.268520  |
| 7 | foo | three | -0.862495 | 0.024580  |

Suppose we wish to compute the standard deviation grouped by the A column. There is a slight problem, namely that we don’t care about the data in column B. We refer to this as a “nuisance” column. If the passed aggregation function can’t be applied to some columns, the troublesome columns will be (silently) dropped. Thus, this does not pose any problems:

```
In [148]: df.groupby('A').std()
Out[148]:
```

|     | C        | D        |
|-----|----------|----------|
| A   |          |          |
| bar | 0.181231 | 1.366330 |
| foo | 0.912265 | 0.884785 |

Note that `df.groupby('A').colname.std().` is more efficient than `df.groupby('A').std().colname`, so if the result of an aggregation function is only interesting over one column (here `colname`), it may be filtered *before* applying the aggregation function.

### 16.9.2 Handling of (un)observed Categorical values

When using a Categorical grouper (as a single grouper, or as part of multiplier groupers), the `observed` keyword controls whether to return a cartesian product of all possible groupers values (`observed=False`) or only those that are observed groupers (`observed=True`).

Show all values:

```
In [149]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'], categories=['a', 'b'], observed=False)).count()
Out[149]:
```

|   |   |
|---|---|
| a | 3 |
| b | 0 |

dtype: int64

Show only the observed values:

```
In [150]: pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'], categories=['a', 'b'], observed=True)).count()
Out[150]:
```

(continues on next page)

(continued from previous page)

```
a      3
dtype: int64
```

The returned dtype of the grouped will *always* include *all* of the categories that were grouped.

```
In [151]: s = pd.Series([1, 1, 1]).groupby(pd.Categorical(['a', 'a', 'a'],
↳categories=['a', 'b']), observed=False).count()

In [152]: s.index.dtype
Out[152]: CategoricalDtype(categories=['a', 'b'], ordered=False)
```

### 16.9.3 NA and NaT group handling

If there are any NaN or NaT values in the grouping key, these will be automatically excluded. In other words, there will never be an “NA group” or “NaT group”. This was not the case in older versions of pandas, but users were generally discarding the NA group anyway (and supporting it was an implementation headache).

### 16.9.4 Grouping with ordered factors

Categorical variables represented as instance of pandas’s `Categorical` class can be used as group keys. If so, the order of the levels will be preserved:

```
In [153]: data = pd.Series(np.random.randn(100))

In [154]: factor = pd.qcut(data, [0, .25, .5, .75, 1.])

In [155]: data.groupby(factor).mean()
Out[155]:
(-2.618, -0.684]    -1.331461
(-0.684, -0.0232]   -0.272816
(-0.0232, 0.541]    0.263607
(0.541, 2.369]      1.166038
dtype: float64
```

### 16.9.5 Grouping with a Grouper specification

You may need to specify a bit more data to properly group. You can use the `pd.Grouper` to provide this local control.

```
In [156]: import datetime

In [157]: df = pd.DataFrame({
.....:     'Branch' : 'A A A A A A B'.split(),
.....:     'Buyer' : 'Carl Mark Carl Carl Joe Joe Carl'.split(),
.....:     'Quantity': [1,3,5,1,8,1,9,3],
.....:     'Date' : [
.....:         datetime.datetime(2013,1,1,13,0),
.....:         datetime.datetime(2013,1,1,13,5),
.....:         datetime.datetime(2013,10,1,20,0),
.....:         datetime.datetime(2013,10,2,10,0),
.....:         datetime.datetime(2013,10,1,20,0),
```

(continues on next page)

(continued from previous page)

```
.....: datetime.datetime(2013,10,2,10,0),
.....: datetime.datetime(2013,12,2,12,0),
.....: datetime.datetime(2013,12,2,14,0),
.....: ]
.....: })
.....:
```

**In [158]:** df

**Out [158]:**

|   | Branch | Buyer | Quantity | Date                |
|---|--------|-------|----------|---------------------|
| 0 | A      | Carl  | 1        | 2013-01-01 13:00:00 |
| 1 | A      | Mark  | 3        | 2013-01-01 13:05:00 |
| 2 | A      | Carl  | 5        | 2013-10-01 20:00:00 |
| 3 | A      | Carl  | 1        | 2013-10-02 10:00:00 |
| 4 | A      | Joe   | 8        | 2013-10-01 20:00:00 |
| 5 | A      | Joe   | 1        | 2013-10-02 10:00:00 |
| 6 | A      | Joe   | 9        | 2013-12-02 12:00:00 |
| 7 | B      | Carl  | 3        | 2013-12-02 14:00:00 |

Groupby a specific column with the desired frequency. This is like resampling.

```
In [159]: df.groupby([pd.Grouper(freq='1M',key='Date'),'Buyer']).sum()
Out[159]:
```

| Date       | Buyer | Quantity |
|------------|-------|----------|
| 2013-01-31 | Carl  | 1        |
|            | Mark  | 3        |
| 2013-10-31 | Carl  | 6        |
|            | Joe   | 9        |
| 2013-12-31 | Carl  | 3        |
|            | Joe   | 9        |

You have an ambiguous specification in that you have a named index and a column that could be potential groupers.

```
In [160]: df = df.set_index('Date')

In [161]: df['Date'] = df.index + pd.offsets.MonthEnd(2)

In [162]: df.groupby([pd.Grouper(freq='6M',key='Date'),'Buyer']).sum()
Out[162]:
```

| Date       | Buyer | Quantity |
|------------|-------|----------|
| 2013-02-28 | Carl  | 1        |
|            | Mark  | 3        |
| 2014-02-28 | Carl  | 9        |
|            | Joe   | 18       |

```
In [163]: df.groupby([pd.Grouper(freq='6M',level='Date'),'Buyer']).sum()
////////////////////////////////////
```

↩

| Date       | Buyer | Quantity |
|------------|-------|----------|
| 2013-01-31 | Carl  | 1        |
|            | Mark  | 3        |
| 2014-01-31 | Carl  | 9        |
|            | Joe   | 18       |



```
In [164]: df = pd.DataFrame([[1, 2], [1, 4], [5, 6]], columns=['A', 'B'])
```

```
In [165]: df
```

Out [165] :

|   | A | B |
|---|---|---|
| 0 | 1 | 2 |
| 1 | 1 | 4 |
| 2 | 5 | 6 |

```
In [166]: g = df.groupby('A')
```

```
In [167]: q.head(1)
```

Out [167] :

|   | A | B |
|---|---|---|
| 0 | 1 | 2 |
| 2 | 5 | 6 |

```
In [168]: q.tail(1)
```

```
Out[168]:
```

|   | A | B |
|---|---|---|
| 1 | 1 | 4 |
| 2 | 5 | 6 |

### 16.9.7 Taking the nth row of each group

```
In [169]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])
```

```
In [170]: g = df.groupby('A')
```

```
In [171]: g.nth(0)
```

Out [171] :

|   | B   |
|---|-----|
| A |     |
| 1 | NaN |
| 5 | 6.0 |

```
In [172]: g.nth(-1)
```

```
Out[172]:
```

| A | B   |
|---|-----|
| 1 | 4.0 |
| 5 | 6.0 |

```
In [173]: g.nth(1)
```

↔ B

## 16.9. Other useful features

(continued from previous page)

```
A
1  4.0
```

If you want to select the *nth* not-null item, use the `dropna` kwarg. For a `DataFrame` this should be either `'any'` or `'all'` just like you would pass to `dropna`:

```
# nth(0) is the same as g.first()
In [174]: g.nth(0, dropna='any')
Out[174]:
      B
A
1  4.0
5  6.0

In [175]: g.first()
Out[175]:
      B
A
1  4.0
5  6.0

# nth(-1) is the same as g.last()
In [176]: g.nth(-1, dropna='any') # NaNs denote group exhausted when using dropna
Out[176]:
      B
A
1  4.0
5  6.0

In [177]: g.last()
Out[177]:
      B
A
1  4.0
5  6.0

In [178]: g.B.nth(0, dropna='all')
Out[178]:
A
1  4.0
5  6.0
Name: B, dtype: float64
```

As with other methods, passing `as_index=False`, will achieve a filtration, which returns the grouped row.

```
In [179]: df = pd.DataFrame([[1, np.nan], [1, 4], [5, 6]], columns=['A', 'B'])
In [180]: g = df.groupby('A', as_index=False)
In [181]: g.nth(0)
Out[181]:
   A  B
0  1 NaN
```

(continues on next page)

(continued from previous page)

```
In [182]: g.nth(-1)
\\Out[182]:
```

|   | A | B   |
|---|---|-----|
| 1 | 1 | 4.0 |
| 2 | 5 | 6.0 |

You can also select multiple rows from each group by specifying multiple `nth` values as a list of ints.

```
In [183]: business_dates = pd.date_range(start='4/1/2014', end='6/30/2014', freq='B')

In [184]: df = pd.DataFrame(1, index=business_dates, columns=['a', 'b'])

# get the first, 4th, and last date index for each month
In [185]: df.groupby([df.index.year, df.index.month]).nth([0, 3, -1])
Out[185]:
```

|      |   | a | b |
|------|---|---|---|
| 2014 | 4 | 1 | 1 |
|      | 4 | 1 | 1 |
|      | 4 | 1 | 1 |
|      | 5 | 1 | 1 |
|      | 5 | 1 | 1 |
|      | 5 | 1 | 1 |
|      | 6 | 1 | 1 |
|      | 6 | 1 | 1 |
|      | 6 | 1 | 1 |

### 16.9.8 Enumerate group items

To see the order in which each row appears within its group, use the `cumcount` method:

```
In [186]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])  
  
In [187]: dfg  
Out[187]:  
   A  
0  a  
1  a  
2  a  
3  b  
4  b  
5  a  
  
In [188]: dfg.groupby('A').cumcount()  
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[188]:  
0    0  
1    1  
2    2  
3    0  
4    1  
5    3  
dtype: int64  
  
In [189]: dfg.groupby('A').cumcount(ascending=False)
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
0      3
1      2
2      1
3      1
4      0
5      0
dtype: int64

```

### 16.9.9 Enumerate groups

New in version 0.20.2.

To see the ordering of the groups (as opposed to the order of rows within a group given by `cumcount`) you can use `ngroup()`.

Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the `groupby` object, not the order they are first observed.

```

In [190]: dfg = pd.DataFrame(list('aaabba'), columns=['A'])

In [191]: dfg
Out[191]:
   A
0  a
1  a
2  a
3  b
4  b
5  a

In [192]: dfg.groupby('A').ngroup()
Out[192]:
0      0
1      0
2      0
3      1
4      1
5      0
dtype: int64

In [193]: dfg.groupby('A').ngroup(ascending=False)
Out[193]:
0      1
1      1
2      1
3      0
4      0
5      1
dtype: int64

```

### 16.9.10 Plotting

Groupby also works with some plotting methods. For example, suppose we suspect that some features in a DataFrame may differ by group, in this case, the values in column 1 where the group is “B” are 3 higher on average.

```
In [194]: np.random.seed(1234)

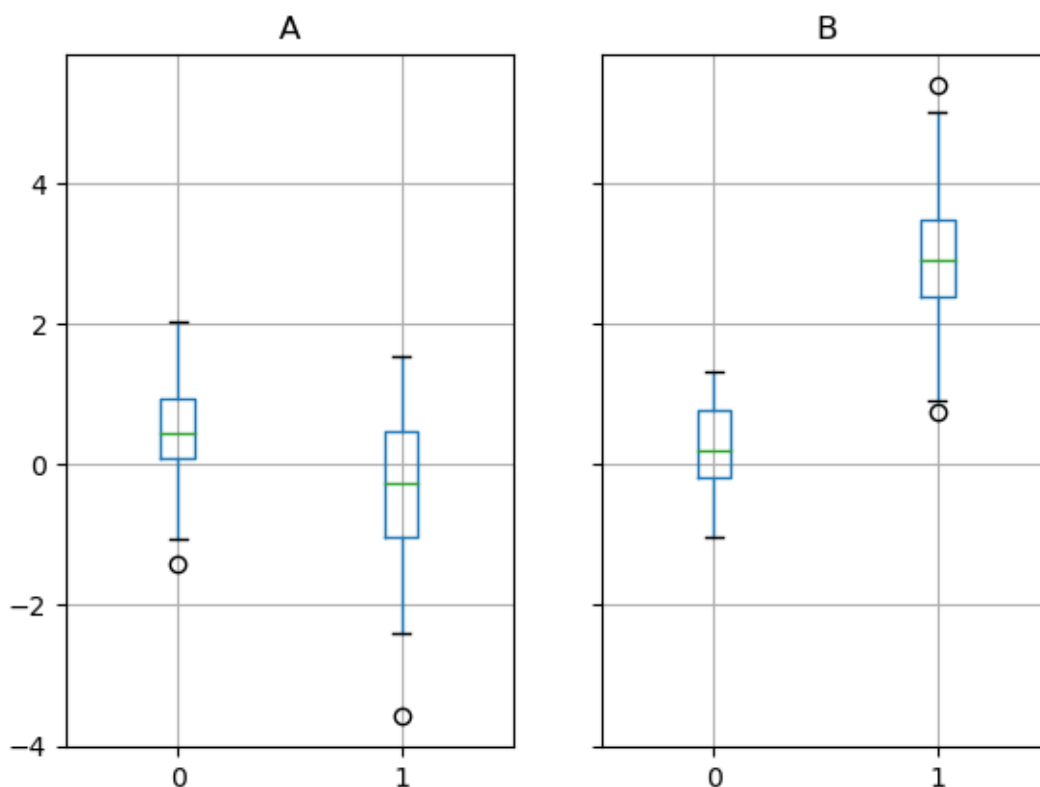
In [195]: df = pd.DataFrame(np.random.randn(50, 2))

In [196]: df['g'] = np.random.choice(['A', 'B'], size=50)

In [197]: df.loc[df['g'] == 'B', 1] += 3
```

We can easily visualize this with a boxplot:

```
In [198]: df.groupby('g').boxplot()
Out[198]:
A      AxesSubplot(0.1,0.15;0.363636x0.75)
B      AxesSubplot(0.536364,0.15;0.363636x0.75)
dtype: object
```



The result of calling `boxplot` is a dictionary whose keys are the values of our grouping column `g` (“A” and “B”). The values of the resulting dictionary can be controlled by the `return_type` keyword of `boxplot`. See the [visualization documentation](#) for more.

**Warning:** For historical reasons, `df.groupby("g").boxplot()` is not equivalent to `df.boxplot(by="g")`. See [here](#) for an explanation.

### 16.9.11 Piping function calls

New in version 0.21.0.

Similar to the functionality provided by `DataFrame` and `Series`, functions that take `GroupBy` objects can be chained together using a `pipe` method to allow for a cleaner, more readable syntax. To read about `.pipe` in general terms, see [here](#).

Combining `.groupby` and `.pipe` is often useful when you need to reuse `GroupBy` objects.

As an example, imagine having a `DataFrame` with columns for stores, products, revenue and quantity sold. We'd like to do a groupwise calculation of *prices* (i.e. revenue/quantity) per store and per product. We could do this in a multi-step operation, but expressing it in terms of piping can make the code more readable. First we set the data:

```
In [199]: import numpy as np

In [200]: n = 1000

In [201]: df = pd.DataFrame({'Store': np.random.choice(['Store_1', 'Store_2'], n),
.....:                      'Product': np.random.choice(['Product_1',
.....:                                                    'Product_2'], n),
.....:                      'Revenue': (np.random.random(n)*50+10).round(2),
.....:                      'Quantity': np.random.randint(1, 10, size=n)})
.....:

In [202]: df.head(2)
Out[202]:
```

|   | Store   | Product   | Revenue | Quantity |
|---|---------|-----------|---------|----------|
| 0 | Store_2 | Product_1 | 26.12   | 1        |
| 1 | Store_2 | Product_1 | 28.86   | 1        |

Now, to find prices per store/product, we can simply do:

```
In [203]: (df.groupby(['Store', 'Product'])
.....:      .pipe(lambda grp: grp.Revenue.sum()/grp.Quantity.sum())
.....:      .unstack().round(2))
.....:
Out[203]:
```

|         | Product_1 | Product_2 |
|---------|-----------|-----------|
| Store   |           |           |
| Store_1 | 6.82      | 7.05      |
| Store_2 | 6.30      | 6.64      |

Piping can also be expressive when you want to deliver a grouped object to some arbitrary function, for example:

```
(df.groupby(['Store', 'Product']).pipe(report_func))
```

where `report_func` takes a `GroupBy` object and creates a report from that.

## 16.10 Examples

### 16.10.1 Regrouping by factor

Regroup columns of a DataFrame according to their sum, and sum the aggregated ones.

```
In [204]: df = pd.DataFrame({'a': [1, 0, 0], 'b': [0, 1, 0], 'c': [1, 0, 0], 'd': [2, 3, 4]})

In [205]: df
Out[205]:
```

|   | a | b | c | d |
|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 2 |
| 1 | 0 | 1 | 0 | 3 |
| 2 | 0 | 0 | 0 | 4 |

```
In [206]: df.groupby(df.sum(), axis=1).sum()
Out[206]:
```

|   | 1 | 9 |
|---|---|---|
| 0 | 2 | 2 |
| 1 | 1 | 3 |
| 2 | 0 | 4 |

### 16.10.2 Multi-column factorization

By using `ngroup()`, we can extract information about the groups in a way similar to `factorize()` (as described further in the [reshaping API](#)) but which applies naturally to multiple columns of mixed type and different sources. This can be useful as an intermediate categorical-like step in processing, when the relationships between the group rows are more important than their content, or as input to an algorithm which only accepts the integer encoding. (For more information about support in pandas for full categorical data, see the [Categorical introduction](#) and the [API documentation](#).)

```
In [207]: dfg = pd.DataFrame({"A": [1, 1, 2, 3, 2], "B": list("aaaba")})

In [208]: dfg
Out[208]:
```

|   | A | B |
|---|---|---|
| 0 | 1 | a |
| 1 | 1 | a |
| 2 | 2 | a |
| 3 | 3 | b |
| 4 | 2 | a |

```
In [209]: dfg.groupby(["A", "B"]).ngroup()
Out[209]:
```

|   | 0 |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 1 |
| 3 | 2 |
| 4 | 1 |

```
dtype: int64

In [210]: dfg.groupby(["A", [0, 0, 0, 1, 1]]).ngroup()
Out[210]:
```

|   | 0 |
|---|---|
| 0 | 0 |

(continues on next page)

(continued from previous page)

```
1      0
2      1
3      3
4      2
dtype: int64
```

### 16.10.3 Groupby by Indexer to ‘resample’ data

Resampling produces new hypothetical samples (resamples) from already existing observed data or from a model that generates data. These new samples are similar to the pre-existing samples.

In order to resample to work on indices that are non-datetime-like, the following procedure can be utilized.

In the following examples, `df.index // 5` returns a binary array which is used to determine what gets selected for the groupby operation.

**Note:** The below example shows how we can downsample by consolidation of samples into fewer samples. Here by using `df.index // 5`, we are aggregating the samples in bins. By applying `std()` function, we aggregate the information contained in many samples into a small subset of values which is their standard deviation thereby reducing the number of samples.

```
In [211]: df = pd.DataFrame(np.random.randn(10,2))
```

```
In [212]: df
```

Out [212] :

|   | 0         | 1         |
|---|-----------|-----------|
| 0 | -0.793893 | 0.321153  |
| 1 | 0.342250  | 1.618906  |
| 2 | -0.975807 | 1.918201  |
| 3 | -0.810847 | -1.405919 |
| 4 | -1.977759 | 0.461659  |
| 5 | 0.730057  | -1.316938 |
| 6 | -0.751328 | 0.528290  |
| 7 | -0.257759 | -1.081009 |
| 8 | 0.505895  | -1.701948 |
| 9 | -1.006349 | 0.020208  |

```
In [213]: df.index // 5
```

[illegible]

```
In [214]: df.groupby(df.index // 5).std()
```

|   | 0        | 1        |
|---|----------|----------|
| 0 | 0.823647 | 1.312912 |
| 1 | 0.760109 | 0.942941 |

#### 16.10.4 Returning a Series to propagate names

Group DataFrame columns, compute a set of metrics and return a named Series. The Series name is used as the name for the column index. This is especially useful in conjunction with reshaping operations such as stacking in which the



column index name will be used as the name of the inserted column:

```
In [215]: df = pd.DataFrame({
.....:     'a':    [0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2],
.....:     'b':    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1],
.....:     'c':    [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0],
.....:     'd':    [0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1],
.....: })

In [216]: def compute_metrics(x):
.....:     result = {'b_sum': x['b'].sum(), 'c_mean': x['c'].mean()}
.....:     return pd.Series(result, name='metrics')
.....:

In [217]: result = df.groupby('a').apply(compute_metrics)

In [218]: result
Out[218]:
```

|   | metrics                           |
|---|-----------------------------------|
| a |                                   |
| 0 | b_sum      2.0<br>c_mean      0.5 |
| 1 | b_sum      2.0<br>c_mean      0.5 |
| 2 | b_sum      2.0<br>c_mean      0.5 |

```
dtype: float64
```



## MERGE, JOIN, AND CONCATENATE

pandas provides various facilities for easily combining together Series, DataFrame, and Panel objects with various kinds of set logic for the indexes and relational algebra functionality in the case of join / merge-type operations.

### 17.1 Concatenating objects

The `concat()` function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes. Note that I say “if any” because there is only a single possible axis of concatenation for Series.

Before diving into all of the details of `concat` and what it can do, here is a simple example:

```
In [1]: df1 = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
...:                      'B': ['B0', 'B1', 'B2', 'B3'],
...:                      'C': ['C0', 'C1', 'C2', 'C3'],
...:                      'D': ['D0', 'D1', 'D2', 'D3']},
...:                      index=[0, 1, 2, 3])

In [2]: df2 = pd.DataFrame({'A': ['A4', 'A5', 'A6', 'A7'],
...:                      'B': ['B4', 'B5', 'B6', 'B7'],
...:                      'C': ['C4', 'C5', 'C6', 'C7'],
...:                      'D': ['D4', 'D5', 'D6', 'D7']},
...:                      index=[4, 5, 6, 7])

In [3]: df3 = pd.DataFrame({'A': ['A8', 'A9', 'A10', 'A11'],
...:                      'B': ['B8', 'B9', 'B10', 'B11'],
...:                      'C': ['C8', 'C9', 'C10', 'C11'],
...:                      'D': ['D8', 'D9', 'D10', 'D11']},
...:                      index=[8, 9, 10, 11])

In [4]: frames = [df1, df2, df3]

In [5]: result = pd.concat(frames)
```

| df1 |     |     |     |     | Result |     |     |     |     |
|-----|-----|-----|-----|-----|--------|-----|-----|-----|-----|
|     | A   | B   | C   | D   |        | A   | B   | C   | D   |
| 0   | A0  | B0  | C0  | D0  | 0      | A0  | B0  | C0  | D0  |
| 1   | A1  | B1  | C1  | D1  | 1      | A1  | B1  | C1  | D1  |
| 2   | A2  | B2  | C2  | D2  | 2      | A2  | B2  | C2  | D2  |
| 3   | A3  | B3  | C3  | D3  | 3      | A3  | B3  | C3  | D3  |
| df2 |     |     |     |     | 4      | A4  | B4  | C4  | D4  |
|     | A   | B   | C   | D   | 5      | A5  | B5  | C5  | D5  |
| 4   | A4  | B4  | C4  | D4  | 6      | A6  | B6  | C6  | D6  |
| 5   | A5  | B5  | C5  | D5  | 7      | A7  | B7  | C7  | D7  |
| 6   | A6  | B6  | C6  | D6  | df3    |     |     |     |     |
| 7   | A7  | B7  | C7  | D7  |        | A   | B   | C   | D   |
| df3 |     |     |     |     | 8      | A8  | B8  | C8  | D8  |
|     | A   | B   | C   | D   | 9      | A9  | B9  | C9  | D9  |
| 8   | A8  | B8  | C8  | D8  | 10     | A10 | B10 | C10 | D10 |
| 9   | A9  | B9  | C9  | D9  | 11     | A11 | B11 | C11 | D11 |
| 10  | A10 | B10 | C10 | D10 |        |     |     |     |     |
| 11  | A11 | B11 | C11 | D11 |        |     |     |     |     |

Like its sibling function on ndarrays, `numpy.concatenate`, `pandas.concat` takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of “what to do with the other axes”:

```
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

- `objs`: a sequence or mapping of Series, DataFrame, or Panel objects. If a dict is passed, the sorted keys will be used as the `keys` argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a `ValueError` will be raised.
- `axis`: {0, 1, ...}, default 0. The axis to concatenate along.
- `join`: {'inner', 'outer'}, default 'outer'. How to handle indexes on other axis(es). Outer for union and inner for intersection.
- `ignore_index`: boolean, default False. If True, do not use the index values on the concatenation axis. The resulting axis will be labeled 0, ..., n - 1. This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.
- `join_axes`: list of Index objects. Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic.
- `keys`: sequence, default None. Construct hierarchical index using the passed keys as the outermost level. If multiple levels passed, should contain tuples.
- `levels`: list of sequences, default None. Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys.
- `names`: list, default None. Names for the levels in the resulting hierarchical index.
- `verify_integrity`: boolean, default False. Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation.

- `copy` : boolean, default True. If False, do not copy data unnecessarily.

Without a little bit of context many of these arguments don't make much sense. Let's revisit the above example. Suppose we wanted to associate specific keys with each of the pieces of the chopped up DataFrame. We can do this using the `keys` argument:

```
In [6]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

| df1 |     |     |     |     | Result |    |     |     |     |     |
|-----|-----|-----|-----|-----|--------|----|-----|-----|-----|-----|
|     | A   | B   | C   | D   |        |    | A   | B   | C   | D   |
| 0   | A0  | B0  | C0  | D0  | x      | 0  | A0  | B0  | C0  | D0  |
| 1   | A1  | B1  | C1  | D1  |        | 1  | A1  | B1  | C1  | D1  |
| 2   | A2  | B2  | C2  | D2  |        | 2  | A2  | B2  | C2  | D2  |
| 3   | A3  | B3  | C3  | D3  |        | 3  | A3  | B3  | C3  | D3  |
| df2 |     |     |     |     | y      | 4  | A4  | B4  | C4  | D4  |
|     | A   | B   | C   | D   |        | 5  | A5  | B5  | C5  | D5  |
| 4   | A4  | B4  | C4  | D4  |        | 6  | A6  | B6  | C6  | D6  |
| 5   | A5  | B5  | C5  | D5  |        | 7  | A7  | B7  | C7  | D7  |
| 6   | A6  | B6  | C6  | D6  | z      | 8  | A8  | B8  | C8  | D8  |
| 7   | A7  | B7  | C7  | D7  |        | 9  | A9  | B9  | C9  | D9  |
| df3 |     |     |     |     |        | 10 | A10 | B10 | C10 | D10 |
|     | A   | B   | C   | D   |        | 11 | A11 | B11 | C11 | D11 |
| 8   | A8  | B8  | C8  | D8  |        |    |     |     |     |     |
| 9   | A9  | B9  | C9  | D9  |        |    |     |     |     |     |
| 10  | A10 | B10 | C10 | D10 |        |    |     |     |     |     |
| 11  | A11 | B11 | C11 | D11 |        |    |     |     |     |     |

As you can see (if you've read the rest of the documentation), the resulting object's index has a *hierarchical index*. This means that we can now select out each chunk by key:

```
In [7]: result.loc['y']
```

```
Out [7]:
```

```
   A  B  C  D
4  A4 B4 C4 D4
5  A5 B5 C5 D5
6  A6 B6 C6 D6
7  A7 B7 C7 D7
```

It's not a stretch to see how this can be very useful. More detail on this functionality below.

**Note:** It is worth noting that `concat()` (and therefore `append()`) makes a full copy of the data, and that constantly reusing this function can create a significant performance hit. If you need to use the operation over several datasets, use a list comprehension.

```
frames = [ process_your_file(f) for f in files ]
result = pd.concat(frames)
```

### 17.1.1 Set logic on the other axes

When gluing together multiple DataFrames, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in the following three ways:

- Take the union of them all, `join='outer'`. This is the default option as it results in zero information loss.
- Take the intersection, `join='inner'`.
- Use a specific index, as passed to the `join_axes` argument.

Here is an example of each of these methods. First, the default `join='outer'` behavior:

```
In [8]: df4 = pd.DataFrame({'B': ['B2', 'B3', 'B6', 'B7'],
...:                       'D': ['D2', 'D3', 'D6', 'D7'],
...:                       'F': ['F2', 'F3', 'F6', 'F7']},
...:                       index=[2, 3, 6, 7])

In [9]: result = pd.concat([df1, df4], axis=1, sort=False)
```

| df1 |    |    |    |    | df4 |    |    |    | Result |     |     |     |     |     |     |     |
|-----|----|----|----|----|-----|----|----|----|--------|-----|-----|-----|-----|-----|-----|-----|
|     | A  | B  | C  | D  |     | B  | D  | F  |        | A   | B   | C   | D   | B   | D   | F   |
| 0   | A0 | B0 | C0 | D0 | 2   | B2 | D2 | F2 | 0      | A0  | B0  | C0  | D0  | NaN | NaN | NaN |
| 1   | A1 | B1 | C1 | D1 | 3   | B3 | D3 | F3 | 1      | A1  | B1  | C1  | D1  | NaN | NaN | NaN |
| 2   | A2 | B2 | C2 | D2 | 6   | B6 | D6 | F6 | 2      | A2  | B2  | C2  | D2  | B2  | D2  | F2  |
| 3   | A3 | B3 | C3 | D3 | 7   | B7 | D7 | F7 | 3      | A3  | B3  | C3  | D3  | B3  | D3  | F3  |
|     |    |    |    |    |     |    |    |    | 6      | NaN | NaN | NaN | NaN | B6  | D6  | F6  |
|     |    |    |    |    |     |    |    |    | 7      | NaN | NaN | NaN | NaN | B7  | D7  | F7  |

**Warning:** Changed in version 0.23.0.

The default behavior with `join='outer'` is to sort the other axis (columns in this case). In a future version of pandas, the default will be to not sort. We specified `sort=False` to opt in to the new behavior now.

Here is the same thing with `join='inner'`:

```
In [10]: result = pd.concat([df1, df4], axis=1, join='inner')
```

| df1 |    |    |    |    | df4 |    |    |    | Result |    |    |    |    |    |    |    |
|-----|----|----|----|----|-----|----|----|----|--------|----|----|----|----|----|----|----|
|     | A  | B  | C  | D  |     | B  | D  | F  |        | A  | B  | C  | D  | B  | D  | F  |
| 0   | A0 | B0 | C0 | D0 | 2   | B2 | D2 | F2 |        |    |    |    |    |    |    |    |
| 1   | A1 | B1 | C1 | D1 | 3   | B3 | D3 | F3 |        |    |    |    |    |    |    |    |
| 2   | A2 | B2 | C2 | D2 | 6   | B6 | D6 | F6 | 2      | A2 | B2 | C2 | D2 | B2 | D2 | F2 |
| 3   | A3 | B3 | C3 | D3 | 7   | B7 | D7 | F7 | 3      | A3 | B3 | C3 | D3 | B3 | D3 | F3 |

Lastly, suppose we just wanted to reuse the *exact index* from the original DataFrame:

```
In [11]: result = pd.concat([df1, df4], axis=1, join_axes=[df1.index])
```

| df1 |    |    |    |    | df4 |    |    |    | Result |    |    |    |    |     |     |     |
|-----|----|----|----|----|-----|----|----|----|--------|----|----|----|----|-----|-----|-----|
|     | A  | B  | C  | D  |     | B  | D  | F  |        | A  | B  | C  | D  | B   | D   | F   |
| 0   | A0 | B0 | C0 | D0 | 2   | B2 | D2 | F2 | 0      | A0 | B0 | C0 | D0 | NaN | NaN | NaN |
| 1   | A1 | B1 | C1 | D1 | 3   | B3 | D3 | F3 | 1      | A1 | B1 | C1 | D1 | NaN | NaN | NaN |
| 2   | A2 | B2 | C2 | D2 | 6   | B6 | D6 | F6 | 2      | A2 | B2 | C2 | D2 | B2  | D2  | F2  |
| 3   | A3 | B3 | C3 | D3 | 7   | B7 | D7 | F7 | 3      | A3 | B3 | C3 | D3 | B3  | D3  | F3  |

### 17.1.2 Concatenating using `append`

A useful shortcut to `concat()` are the `append()` instance methods on Series and DataFrame. These methods actually predated `concat`. They concatenate along `axis=0`, namely the index:

```
In [12]: result = df1.append(df2)
```

| df1 |    |    |    |    | Result |    |    |    |    |
|-----|----|----|----|----|--------|----|----|----|----|
|     | A  | B  | C  | D  |        | A  | B  | C  | D  |
| 0   | A0 | B0 | C0 | D0 | 0      | A0 | B0 | C0 | D0 |
| 1   | A1 | B1 | C1 | D1 | 1      | A1 | B1 | C1 | D1 |
| 2   | A2 | B2 | C2 | D2 | 2      | A2 | B2 | C2 | D2 |
| 3   | A3 | B3 | C3 | D3 | 3      | A3 | B3 | C3 | D3 |
| df2 |    |    |    |    | 4      | A4 | B4 | C4 | D4 |
|     | A  | B  | C  | D  | 5      | A5 | B5 | C5 | D5 |
| 4   | A4 | B4 | C4 | D4 | 6      | A6 | B6 | C6 | D6 |
| 5   | A5 | B5 | C5 | D5 | 7      | A7 | B7 | C7 | D7 |
| 6   | A6 | B6 | C6 | D6 |        |    |    |    |    |
| 7   | A7 | B7 | C7 | D7 |        |    |    |    |    |

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:

```
In [13]: result = df1.append(df4)
```

| df1 |    |    |    |    | Result |     |    |     |    |     |
|-----|----|----|----|----|--------|-----|----|-----|----|-----|
|     | A  | B  | C  | D  |        | A   | B  | C   | D  | F   |
| 0   | A0 | B0 | C0 | D0 | 0      | A0  | B0 | C0  | D0 | NaN |
| 1   | A1 | B1 | C1 | D1 | 1      | A1  | B1 | C1  | D1 | NaN |
| 2   | A2 | B2 | C2 | D2 | 2      | A2  | B2 | C2  | D2 | NaN |
| 3   | A3 | B3 | C3 | D3 | 3      | A3  | B3 | C3  | D3 | NaN |
|     |    |    |    |    | 2      | NaN | B2 | NaN | D2 | F2  |
|     |    |    |    |    | 3      | NaN | B3 | NaN | D3 | F3  |
|     |    |    |    |    | 6      | NaN | B6 | NaN | D6 | F6  |
|     |    |    |    |    | 7      | NaN | B7 | NaN | D7 | F7  |

append may take multiple objects to concatenate:

```
In [14]: result = df1.append([df2, df3])
```

| df1 |    |    |    |    | Result |     |     |     |     |
|-----|----|----|----|----|--------|-----|-----|-----|-----|
|     | A  | B  | C  | D  |        | A   | B   | C   | D   |
| 0   | A0 | B0 | C0 | D0 | 0      | A0  | B0  | C0  | D0  |
| 1   | A1 | B1 | C1 | D1 | 1      | A1  | B1  | C1  | D1  |
| 2   | A2 | B2 | C2 | D2 | 2      | A2  | B2  | C2  | D2  |
| 3   | A3 | B3 | C3 | D3 | 3      | A3  | B3  | C3  | D3  |
|     |    |    |    |    | 4      | A4  | B4  | C4  | D4  |
|     |    |    |    |    | 5      | A5  | B5  | C5  | D5  |
|     |    |    |    |    | 6      | A6  | B6  | C6  | D6  |
|     |    |    |    |    | 7      | A7  | B7  | C7  | D7  |
|     |    |    |    |    | 8      | A8  | B8  | C8  | D8  |
|     |    |    |    |    | 9      | A9  | B9  | C9  | D9  |
|     |    |    |    |    | 10     | A10 | B10 | C10 | D10 |
|     |    |    |    |    | 11     | A11 | B11 | C11 | D11 |

**Note:** Unlike the `append()` method, which appends to the original list and returns `None`, `append()` here **does** not modify `df1` and returns its copy with `df2` appended.



### 17.1.3 Ignoring indexes on the concatenation axis

For `DataFrame`s which don't have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes. To do this, use the `ignore_index` argument:

```
In [15]: result = pd.concat([df1, df4], ignore_index=True)
```

| df1 |    |    |    |    | Result |     |     |     |    |     |
|-----|----|----|----|----|--------|-----|-----|-----|----|-----|
|     | A  | B  | C  | D  |        | A   | B   | C   | D  | F   |
| 0   | A0 | B0 | C0 | D0 | 0      | A0  | B0  | C0  | D0 | NaN |
| 1   | A1 | B1 | C1 | D1 | 1      | A1  | B1  | C1  | D1 | NaN |
| 2   | A2 | B2 | C2 | D2 | 2      | A2  | B2  | C2  | D2 | NaN |
| 3   | A3 | B3 | C3 | D3 | 3      | A3  | B3  | C3  | D3 | NaN |
| df4 |    |    |    |    | 4      | NaN | B2  | NaN | D2 | F2  |
|     | B  | D  | F  |    | 5      | NaN | B3  | NaN | D3 | F3  |
| 2   | B2 | D2 | F2 | 6  | NaN    | B6  | NaN | D6  | F6 |     |
| 3   | B3 | D3 | F3 | 7  | NaN    | B7  | NaN | D7  | F7 |     |
| 6   | B6 | D6 | F6 |    |        |     |     |     |    |     |
| 7   | B7 | D7 | F7 |    |        |     |     |     |    |     |

This is also a valid argument to `DataFrame.append()`:

```
In [16]: result = df1.append(df4, ignore_index=True)
```

| df1 |    |    |    |    | Result |     |     |     |    |     |
|-----|----|----|----|----|--------|-----|-----|-----|----|-----|
|     | A  | B  | C  | D  |        | A   | B   | C   | D  | F   |
| 0   | A0 | B0 | C0 | D0 | 0      | A0  | B0  | C0  | D0 | NaN |
| 1   | A1 | B1 | C1 | D1 | 1      | A1  | B1  | C1  | D1 | NaN |
| 2   | A2 | B2 | C2 | D2 | 2      | A2  | B2  | C2  | D2 | NaN |
| 3   | A3 | B3 | C3 | D3 | 3      | A3  | B3  | C3  | D3 | NaN |
| df4 |    |    |    |    | 4      | NaN | B2  | NaN | D2 | F2  |
|     | B  | D  | F  |    | 5      | NaN | B3  | NaN | D3 | F3  |
| 2   | B2 | D2 | F2 | 6  | NaN    | B6  | NaN | D6  | F6 |     |
| 3   | B3 | D3 | F3 | 7  | NaN    | B7  | NaN | D7  | F7 |     |
| 6   | B6 | D6 | F6 |    |        |     |     |     |    |     |
| 7   | B7 | D7 | F7 |    |        |     |     |     |    |     |

### 17.1.4 Concatenating with mixed ndims

You can concatenate a mix of `Series` and `DataFrame`s. The `Series` will be transformed to `DataFrame` with the column name as the name of the `Series`.

```
In [17]: s1 = pd.Series(['X0', 'X1', 'X2', 'X3'], name='X')
```

```
In [18]: result = pd.concat([df1, s1], axis=1)
```

| df1 |    |    |    |    | s1 |    | Result |    |    |    |    |    |
|-----|----|----|----|----|----|----|--------|----|----|----|----|----|
|     | A  | B  | C  | D  |    | X  |        | A  | B  | C  | D  | X  |
| 0   | A0 | B0 | C0 | D0 | 0  | X0 | 0      | A0 | B0 | C0 | D0 | X0 |
| 1   | A1 | B1 | C1 | D1 | 1  | X1 | 1      | A1 | B1 | C1 | D1 | X1 |
| 2   | A2 | B2 | C2 | D2 | 2  | X2 | 2      | A2 | B2 | C2 | D2 | X2 |
| 3   | A3 | B3 | C3 | D3 | 3  | X3 | 3      | A3 | B3 | C3 | D3 | X3 |

**Note:** Since we're concatenating a Series to a DataFrame, we could have achieved the same result with `DataFrame.assign()`. To concatenate an arbitrary number of pandas objects (DataFrame or Series), use `concat`.

If unnamed Series are passed they will be numbered consecutively.

```
In [19]: s2 = pd.Series(['_0', '_1', '_2', '_3'])
```

```
In [20]: result = pd.concat([df1, s2, s2, s2], axis=1)
```

| df1 |    |    |    |    | s2 |    | Result |    |    |    |    |    |    |    |
|-----|----|----|----|----|----|----|--------|----|----|----|----|----|----|----|
|     | A  | B  | C  | D  |    |    |        | A  | B  | C  | D  | 0  | 1  | 2  |
| 0   | A0 | B0 | C0 | D0 | 0  | _0 | 0      | A0 | B0 | C0 | D0 | _0 | _0 | _0 |
| 1   | A1 | B1 | C1 | D1 | 1  | _1 | 1      | A1 | B1 | C1 | D1 | _1 | _1 | _1 |
| 2   | A2 | B2 | C2 | D2 | 2  | _2 | 2      | A2 | B2 | C2 | D2 | _2 | _2 | _2 |
| 3   | A3 | B3 | C3 | D3 | 3  | _3 | 3      | A3 | B3 | C3 | D3 | _3 | _3 | _3 |

Passing `ignore_index=True` will drop all name references.

```
In [21]: result = pd.concat([df1, s1], axis=1, ignore_index=True)
```

| df1 |    |    |    |    | s1 |    | Result |    |    |    |    |    |
|-----|----|----|----|----|----|----|--------|----|----|----|----|----|
|     | A  | B  | C  | D  |    | X  |        | 0  | 1  | 2  | 3  | 4  |
| 0   | A0 | B0 | C0 | D0 | 0  | X0 | 0      | A0 | B0 | C0 | D0 | X0 |
| 1   | A1 | B1 | C1 | D1 | 1  | X1 | 1      | A1 | B1 | C1 | D1 | X1 |
| 2   | A2 | B2 | C2 | D2 | 2  | X2 | 2      | A2 | B2 | C2 | D2 | X2 |
| 3   | A3 | B3 | C3 | D3 | 3  | X3 | 3      | A3 | B3 | C3 | D3 | X3 |

### 17.1.5 More concatenating with group keys

A fairly common use of the `keys` argument is to override the column names when creating a new `DataFrame` based on existing `Series`. Notice how the default behaviour consists on letting the resulting `DataFrame` inherit the parent `Series`' name, when these existed.

```
In [22]: s3 = pd.Series([0, 1, 2, 3], name='foo')
```

```
In [23]: s4 = pd.Series([0, 1, 2, 3])
```

```
In [24]: s5 = pd.Series([0, 1, 4, 5])
```

```
In [25]: pd.concat([s3, s4, s5], axis=1)
```

```
Out[25]:
   foo  0  1
0    0  0  0
1    1  1  1
2    2  2  4
3    3  3  5
```

Through the `keys` argument we can override the existing column names.

```
In [26]: pd.concat([s3, s4, s5], axis=1, keys=['red', 'blue', 'yellow'])
```

```
Out[26]:
   red  blue  yellow
0    0    0    0
1    1    1    1
2    2    2    4
3    3    3    5
```

Let's consider a variation of the very first example presented:

```
In [27]: result = pd.concat(frames, keys=['x', 'y', 'z'])
```

| df1 |     |     |     |     | Result |    |     |     |     |     |
|-----|-----|-----|-----|-----|--------|----|-----|-----|-----|-----|
|     | A   | B   | C   | D   |        |    | A   | B   | C   | D   |
| 0   | A0  | B0  | C0  | D0  | x      | 0  | A0  | B0  | C0  | D0  |
| 1   | A1  | B1  | C1  | D1  |        | 1  | A1  | B1  | C1  | D1  |
| 2   | A2  | B2  | C2  | D2  |        | 2  | A2  | B2  | C2  | D2  |
| 3   | A3  | B3  | C3  | D3  |        | 3  | A3  | B3  | C3  | D3  |
| df2 |     |     |     |     | y      | 4  | A4  | B4  | C4  | D4  |
|     | A   | B   | C   | D   | y      | 5  | A5  | B5  | C5  | D5  |
| 4   | A4  | B4  | C4  | D4  | y      | 6  | A6  | B6  | C6  | D6  |
| 5   | A5  | B5  | C5  | D5  | y      | 7  | A7  | B7  | C7  | D7  |
| 6   | A6  | B6  | C6  | D6  | z      | 8  | A8  | B8  | C8  | D8  |
| 7   | A7  | B7  | C7  | D7  | z      | 9  | A9  | B9  | C9  | D9  |
| df3 |     |     |     |     | z      | 10 | A10 | B10 | C10 | D10 |
|     | A   | B   | C   | D   | z      | 11 | A11 | B11 | C11 | D11 |
| 8   | A8  | B8  | C8  | D8  |        |    |     |     |     |     |
| 9   | A9  | B9  | C9  | D9  |        |    |     |     |     |     |
| 10  | A10 | B10 | C10 | D10 |        |    |     |     |     |     |
| 11  | A11 | B11 | C11 | D11 |        |    |     |     |     |     |

You can also pass a dict to `concat` in which case the dict keys will be used for the `keys` argument (unless other keys are specified):

```
In [28]: pieces = {'x': df1, 'y': df2, 'z': df3}
```

```
In [29]: result = pd.concat(pieces)
```

| df1 |     |     |     |     | Result |    |     |     |     |     |
|-----|-----|-----|-----|-----|--------|----|-----|-----|-----|-----|
|     | A   | B   | C   | D   |        |    | A   | B   | C   | D   |
| 0   | A0  | B0  | C0  | D0  | x      | 0  | A0  | B0  | C0  | D0  |
| 1   | A1  | B1  | C1  | D1  |        | 1  | A1  | B1  | C1  | D1  |
| 2   | A2  | B2  | C2  | D2  |        | 2  | A2  | B2  | C2  | D2  |
| 3   | A3  | B3  | C3  | D3  |        | 3  | A3  | B3  | C3  | D3  |
| df2 |     |     |     |     | y      | 4  | A4  | B4  | C4  | D4  |
|     | A   | B   | C   | D   |        | 5  | A5  | B5  | C5  | D5  |
| 4   | A4  | B4  | C4  | D4  |        | 6  | A6  | B6  | C6  | D6  |
| 5   | A5  | B5  | C5  | D5  |        | 7  | A7  | B7  | C7  | D7  |
| 6   | A6  | B6  | C6  | D6  | z      | 8  | A8  | B8  | C8  | D8  |
| 7   | A7  | B7  | C7  | D7  |        | 9  | A9  | B9  | C9  | D9  |
| df3 |     |     |     |     |        | 10 | A10 | B10 | C10 | D10 |
|     | A   | B   | C   | D   |        | 11 | A11 | B11 | C11 | D11 |
| 8   | A8  | B8  | C8  | D8  |        |    |     |     |     |     |
| 9   | A9  | B9  | C9  | D9  |        |    |     |     |     |     |
| 10  | A10 | B10 | C10 | D10 |        |    |     |     |     |     |
| 11  | A11 | B11 | C11 | D11 |        |    |     |     |     |     |

```
In [30]: result = pd.concat(pieces, keys=['z', 'y'])
```

| df1 |     |     |     |     | Result |   |     |     |     |     |
|-----|-----|-----|-----|-----|--------|---|-----|-----|-----|-----|
|     | A   | B   | C   | D   |        |   | A   | B   | C   | D   |
| 0   | A0  | B0  | C0  | D0  | z      | 8 | A8  | B8  | C8  | D8  |
| 1   | A1  | B1  | C1  | D1  |        |   | A9  | B9  | C9  | D9  |
| 2   | A2  | B2  | C2  | D2  |        |   | A10 | B10 | C10 | D10 |
| 3   | A3  | B3  | C3  | D3  |        |   | A11 | B11 | C11 | D11 |
| df2 |     |     |     |     | y      | 4 | A4  | B4  | C4  | D4  |
|     | A   | B   | C   | D   |        |   | A5  | B5  | C5  | D5  |
| 4   | A4  | B4  | C4  | D4  |        |   | A6  | B6  | C6  | D6  |
| 5   | A5  | B5  | C5  | D5  |        |   | A7  | B7  | C7  | D7  |
| 6   | A6  | B6  | C6  | D6  | y      | 5 | A5  | B5  | C5  | D5  |
| 7   | A7  | B7  | C7  | D7  |        |   | A6  | B6  | C6  | D6  |
| df3 |     |     |     |     |        |   | A7  | B7  | C7  | D7  |
|     | A   | B   | C   | D   |        |   |     |     |     |     |
| 8   | A8  | B8  | C8  | D8  | y      | 6 | A6  | B6  | C6  | D6  |
| 9   | A9  | B9  | C9  | D9  |        |   | A7  | B7  | C7  | D7  |
| 10  | A10 | B10 | C10 | D10 |        |   |     |     |     |     |
| 11  | A11 | B11 | C11 | D11 |        |   |     |     |     |     |

The MultiIndex created has levels that are constructed from the passed keys and the index of the DataFrame pieces:

```
In [31]: result.index.levels
Out[31]: FrozenList([[ 'z', 'y' ], [4, 5, 6, 7, 8, 9, 10, 11]])
```

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

```
In [32]: result = pd.concat(pieces, keys=[ 'x', 'y', 'z' ],
.....:                      levels=[ 'z', 'y', 'x', 'w' ],
.....:                      names=[ 'group_key' ])
.....:
```

| df1 |     |     |     |     | Result |    |     |     |     |     |
|-----|-----|-----|-----|-----|--------|----|-----|-----|-----|-----|
|     | A   | B   | C   | D   |        |    | A   | B   | C   | D   |
| 0   | A0  | B0  | C0  | D0  | x      | 0  | A0  | B0  | C0  | D0  |
| 1   | A1  | B1  | C1  | D1  |        | 1  | A1  | B1  | C1  | D1  |
| 2   | A2  | B2  | C2  | D2  |        | 2  | A2  | B2  | C2  | D2  |
| 3   | A3  | B3  | C3  | D3  |        | 3  | A3  | B3  | C3  | D3  |
| df2 |     |     |     |     | y      | 4  | A4  | B4  | C4  | D4  |
|     | A   | B   | C   | D   |        | 5  | A5  | B5  | C5  | D5  |
| 4   | A4  | B4  | C4  | D4  |        | 6  | A6  | B6  | C6  | D6  |
| 5   | A5  | B5  | C5  | D5  |        | 7  | A7  | B7  | C7  | D7  |
| 6   | A6  | B6  | C6  | D6  | z      | 8  | A8  | B8  | C8  | D8  |
| 7   | A7  | B7  | C7  | D7  |        | 9  | A9  | B9  | C9  | D9  |
| df3 |     |     |     |     |        | 10 | A10 | B10 | C10 | D10 |
|     | A   | B   | C   | D   |        | 11 | A11 | B11 | C11 | D11 |
| 8   | A8  | B8  | C8  | D8  |        |    |     |     |     |     |
| 9   | A9  | B9  | C9  | D9  |        |    |     |     |     |     |
| 10  | A10 | B10 | C10 | D10 |        |    |     |     |     |     |
| 11  | A11 | B11 | C11 | D11 |        |    |     |     |     |     |

```
In [33]: result.index.levels
Out[33]: FrozenList([[ 'z', 'y', 'x', 'w' ], [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]])
```

This is fairly esoteric, but it is actually necessary for implementing things like `GroupBy` where the order of a categorical variable is meaningful.

### 17.1.6 Appending rows to a DataFrame

While not especially efficient (since a new object must be created), you can append a single row to a `DataFrame` by passing a `Series` or dict to `append`, which returns a new `DataFrame` as above.

```
In [34]: s2 = pd.Series([ 'X0', 'X1', 'X2', 'X3' ], index=[ 'A', 'B', 'C', 'D' ])
In [35]: result = df1.append(s2, ignore_index=True)
```

| df1 |    |    |    |    | Result |    |    |    |    |
|-----|----|----|----|----|--------|----|----|----|----|
|     | A  | B  | C  | D  |        | A  | B  | C  | D  |
| 0   | A0 | B0 | C0 | D0 | 0      | A0 | B0 | C0 | D0 |
| 1   | A1 | B1 | C1 | D1 | 1      | A1 | B1 | C1 | D1 |
| 2   | A2 | B2 | C2 | D2 | 2      | A2 | B2 | C2 | D2 |
| 3   | A3 | B3 | C3 | D3 | 3      | A3 | B3 | C3 | D3 |

| s2 |   |  |  |    |   |    |    |    |    |
|----|---|--|--|----|---|----|----|----|----|
|    | A |  |  |    |   |    |    |    |    |
|    |   |  |  | X0 | 4 | X0 | X1 | X2 | X3 |
| A  |   |  |  | X0 |   |    |    |    |    |
| B  |   |  |  | X1 |   |    |    |    |    |
| C  |   |  |  | X2 |   |    |    |    |    |
| D  |   |  |  | X3 |   |    |    |    |    |

You should use `ignore_index` with this method to instruct `DataFrame` to discard its index. If you wish to preserve the index, you should construct an appropriately-indexed `DataFrame` and append or concatenate those objects.

You can also pass a list of dicts or `Series`:

```
In [36]: dicts = [{'A': 1, 'B': 2, 'C': 3, 'X': 4},
.....:           {'A': 5, 'B': 6, 'C': 7, 'Y': 8}]
.....:

In [37]: result = df1.append(dicts, ignore_index=True)
```

| df1 |       |    |    |     | Result |   |    |    |    |     |     |     |
|-----|-------|----|----|-----|--------|---|----|----|----|-----|-----|-----|
|     | A     | B  | C  | D   |        | A | B  | C  | D  | X   | Y   |     |
| 0   | A0    | B0 | C0 | D0  |        | 0 | A0 | B0 | C0 | D0  | NaN | NaN |
| 1   | A1    | B1 | C1 | D1  |        | 1 | A1 | B1 | C1 | D1  | NaN | NaN |
| 2   | A2    | B2 | C2 | D2  |        | 2 | A2 | B2 | C2 | D2  | NaN | NaN |
| 3   | A3    | B3 | C3 | D3  |        | 3 | A3 | B3 | C3 | D3  | NaN | NaN |
|     | dicts |    |    |     |        | 4 | 1  | 2  | 3  | NaN | 4.0 | NaN |
|     | A     | B  | C  | X   | Y      | 5 | 5  | 6  | 7  | NaN | 8.0 |     |
| 0   | 1     | 2  | 3  | 4.0 | NaN    |   |    |    |    |     |     |     |
| 1   | 5     | 6  | 7  | NaN | 8.0    |   |    |    |    |     |     |     |

## 17.2 Database-style DataFrame joining/merging

pandas has full-featured, **high performance** in-memory join operations idiomatically very similar to relational databases like SQL. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like `base::merge.data.frame` in R). The reason for this is careful algorithmic design and the internal layout of the data in `DataFrame`.

See the [cookbook](#) for some advanced strategies.

Users who are familiar with SQL but new to pandas might be interested in a [comparison with SQL](#).

pandas provides a single function, `merge()`, as the entry point for all standard database join operations between `DataFrame` objects:

```
pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None,
         left_index=False, right_index=False, sort=True,
         suffixes=('_x', '_y'), copy=True, indicator=False,
         validate=None)
```

- `left`: A `DataFrame` object.
- `right`: Another `DataFrame` object.
- `on`: Column or index level names to join on. Must be found in both the left and right `DataFrame` objects. If not passed and `left_index` and `right_index` are `False`, the intersection of the columns in the `DataFrames` will be inferred to be the join keys.
- `left_on`: Columns or index levels from the left `DataFrame` to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the `DataFrame`.
- `right_on`: Columns or index levels from the right `DataFrame` to use as keys. Can either be column names, index level names, or arrays with length equal to the length of the `DataFrame`.
- `left_index`: If `True`, use the index (row labels) from the left `DataFrame` as its join key(s). In the case of a `DataFrame` with a `MultiIndex` (hierarchical), the number of levels must match the number of join keys from the right `DataFrame`.
- `right_index`: Same usage as `left_index` for the right `DataFrame`.
- `how`: One of `'left'`, `'right'`, `'outer'`, `'inner'`. Defaults to `inner`. See below for more detailed description of each method.
- `sort`: Sort the result `DataFrame` by the join keys in lexicographical order. Defaults to `True`, setting to `False` will improve performance substantially in many cases.
- `suffixes`: A tuple of string suffixes to apply to overlapping columns. Defaults to `('_x', '_y')`.
- `copy`: Always copy data (default `True`) from the passed `DataFrame` objects, even when reindexing is not necessary. Cannot be avoided in many cases but may improve performance / memory usage. The cases where copying can be avoided are somewhat pathological but this option is provided nonetheless.
- `indicator`: Add a column to the output `DataFrame` called `_merge` with information on the source of each row. `_merge` is Categorical-type and takes on a value of `left_only` for observations whose merge key only appears in `'left'` `DataFrame`, `right_only` for observations whose merge key only appears in `'right'` `DataFrame`, and `both` if the observation's merge key is found in both.
- `validate`: string, default `None`. If specified, checks if merge is of specified type.
  - `"one_to_one"` or `"1:1"`: checks if merge keys are unique in both left and right datasets.
  - `"one_to_many"` or `"1:m"`: checks if merge keys are unique in left dataset.
  - `"many_to_one"` or `"m:1"`: checks if merge keys are unique in right dataset.
  - `"many_to_many"` or `"m:m"`: allowed, but does not result in checks.

New in version 0.21.0.

---

**Note:** Support for specifying index levels as the `on`, `left_on`, and `right_on` parameters was added in version 0.23.0.

---

The return type will be the same as `left`. If `left` is a `DataFrame` and `right` is a subclass of `DataFrame`, the return type will still be `DataFrame`.

`merge` is a function in the pandas namespace, and it is also available as a `DataFrame` instance method `merge()`, with the calling `DataFrame` being implicitly considered the left object in the join.



The related `join()` method, uses `merge` internally for the index-on-index (by default) and column(s)-on-index join. If you are joining on index only, you may wish to use `DataFrame.join` to save yourself some typing.

### 17.2.1 Brief primer on merge methods (relational algebra)

Experienced users of relational databases like SQL will be familiar with the terminology used to describe join operations between two SQL-table like structures (`DataFrame` objects). There are several cases to consider which are very important to understand:

- **one-to-one** joins: for example when joining two `DataFrame` objects on their indexes (which must contain unique values).
- **many-to-one** joins: for example when joining an index (unique) to one or more columns in a different `DataFrame`.
- **many-to-many** joins: joining columns on columns.

**Note:** When joining columns on columns (potentially a many-to-many join), any indexes on the passed `DataFrame` objects **will be discarded**.

It is worth spending some time understanding the result of the **many-to-many** join case. In SQL / standard relational algebra, if a key combination appears more than once in both tables, the resulting table will have the **Cartesian product** of the associated data. Here is a very basic example with one unique key combination:

```
In [38]: left = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
.....:                       'A': ['A0', 'A1', 'A2', 'A3'],
.....:                       'B': ['B0', 'B1', 'B2', 'B3']})
.....:

In [39]: right = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3'],
.....:                         'C': ['C0', 'C1', 'C2', 'C3'],
.....:                         'D': ['D0', 'D1', 'D2', 'D3']})
.....:

In [40]: result = pd.merge(left, right, on='key')
```

| left |     |    |    | right |     |    |    | Result |     |    |    |    |    |
|------|-----|----|----|-------|-----|----|----|--------|-----|----|----|----|----|
|      | key | A  | B  |       | key | C  | D  |        | key | A  | B  | C  | D  |
| 0    | K0  | A0 | B0 | 0     | K0  | C0 | D0 | 0      | K0  | A0 | B0 | C0 | D0 |
| 1    | K1  | A1 | B1 | 1     | K1  | C1 | D1 | 1      | K1  | A1 | B1 | C1 | D1 |
| 2    | K2  | A2 | B2 | 2     | K2  | C2 | D2 | 2      | K2  | A2 | B2 | C2 | D2 |
| 3    | K3  | A3 | B3 | 3     | K3  | C3 | D3 | 3      | K3  | A3 | B3 | C3 | D3 |

Here is a more complicated example with multiple join keys. Only the keys appearing in `left` and `right` are present (the intersection), since `how='inner'` by default.

```
In [41]: left = pd.DataFrame({'key1': ['K0', 'K0', 'K1', 'K2'],
.....:                       'key2': ['K0', 'K1', 'K0', 'K1'],
.....:                       'A': ['A0', 'A1', 'A2', 'A3'],
.....:                       'B': ['B0', 'B1', 'B2', 'B3']})
.....:
```

(continues on next page)

(continued from previous page)

```
In [42]: right = pd.DataFrame({'key1': ['K0', 'K1', 'K1', 'K2'],
.....:                       'key2': ['K0', 'K0', 'K0', 'K0'],
.....:                       'C': ['C0', 'C1', 'C2', 'C3'],
.....:                       'D': ['D0', 'D1', 'D2', 'D3']})
.....:

In [43]: result = pd.merge(left, right, on=['key1', 'key2'])
```

| left |      |      |    |    | right |      |      |    |    | Result |      |      |    |    |    |    |
|------|------|------|----|----|-------|------|------|----|----|--------|------|------|----|----|----|----|
|      | key1 | key2 | A  | B  |       | key1 | key2 | C  | D  |        | key1 | key2 | A  | B  | C  | D  |
| 0    | K0   | K0   | A0 | B0 | 0     | K0   | K0   | C0 | D0 | 0      | K0   | K0   | A0 | B0 | C0 | D0 |
| 1    | K0   | K1   | A1 | B1 | 1     | K1   | K0   | C1 | D1 | 1      | K1   | K0   | A2 | B2 | C1 | D1 |
| 2    | K1   | K0   | A2 | B2 | 2     | K1   | K0   | C2 | D2 | 2      | K1   | K0   | A2 | B2 | C2 | D2 |
| 3    | K2   | K1   | A3 | B3 | 3     | K2   | K0   | C3 | D3 |        |      |      |    |    |    |    |

The `how` argument to `merge` specifies how to determine which keys are to be included in the resulting table. If a key combination **does not appear** in either the left or right tables, the values in the joined table will be NA. Here is a summary of the `how` options and their SQL equivalent names:

| Merge method | SQL Join Name    | Description                               |
|--------------|------------------|---|
| left         | LEFT OUTER JOIN  | Use keys from left frame only             |
| right        | RIGHT OUTER JOIN | Use keys from right frame only            |
| outer        | FULL OUTER JOIN  | Use union of keys from both frames        |
| inner        | INNER JOIN       | Use intersection of keys from both frames |

```
In [44]: result = pd.merge(left, right, how='left', on=['key1', 'key2'])
```

| left |      |      |    |    | right |      |      |    |    | Result |      |      |    |    |     |     |
|------|------|------|----|----|-------|------|------|----|----|--------|------|------|----|----|-----|-----|
|      | key1 | key2 | A  | B  |       | key1 | key2 | C  | D  |        | key1 | key2 | A  | B  | C   | D   |
| 0    | K0   | K0   | A0 | B0 | 0     | K0   | K0   | C0 | D0 | 0      | K0   | K0   | A0 | B0 | C0  | D0  |
| 1    | K0   | K1   | A1 | B1 | 1     | K1   | K0   | C1 | D1 | 1      | K0   | K1   | A1 | B1 | NaN | NaN |
| 2    | K1   | K0   | A2 | B2 | 2     | K1   | K0   | C2 | D2 | 2      | K1   | K0   | A2 | B2 | C1  | D1  |
| 3    | K2   | K1   | A3 | B3 | 3     | K2   | K0   | C3 | D3 | 3      | K1   | K0   | A2 | B2 | C2  | D2  |
|      |      |      |    |    |       |      |      |    |    | 4      | K2   | K1   | A3 | B3 | NaN | NaN |

```
In [45]: result = pd.merge(left, right, how='right', on=['key1', 'key2'])
```

| left |      |      |    |    | right |      |      |    |    | Result |      |      |     |     |    |    |
|------|------|------|----|----|-------|------|------|----|----|--------|------|------|-----|-----|----|----|
|      | key1 | key2 | A  | B  |       | key1 | key2 | C  | D  |        | key1 | key2 | A   | B   | C  | D  |
| 0    | K0   | K0   | A0 | B0 | 0     | K0   | K0   | C0 | D0 | 0      | K0   | K0   | A0  | B0  | C0 | D0 |
| 1    | K0   | K1   | A1 | B1 | 1     | K1   | K0   | C1 | D1 | 1      | K1   | K0   | A2  | B2  | C1 | D1 |
| 2    | K1   | K0   | A2 | B2 | 2     | K1   | K0   | C2 | D2 | 2      | K1   | K0   | A2  | B2  | C2 | D2 |
| 3    | K2   | K1   | A3 | B3 | 3     | K2   | K0   | C3 | D3 | 3      | K2   | K0   | NaN | NaN | C3 | D3 |

```
In [46]: result = pd.merge(left, right, how='outer', on=['key1', 'key2'])
```

| left |      |      |    |    | right |      |      |    |    | Result |      |      |     |     |     |     |
|------|------|------|----|----|-------|------|------|----|----|--------|------|------|-----|-----|-----|-----|
|      |      |      |    |    |       |      |      |    |    |        |      |      |     |     |     |     |
|      |      |      |    |    |       |      |      |    |    |        | key1 | key2 | A   | B   | C   | D   |
|      | key1 | key2 | A  | B  |       | key1 | key2 | C  | D  | 0      | K0   | K0   | A0  | B0  | C0  | D0  |
| 0    | K0   | K0   | A0 | B0 | 0     | K0   | K0   | C0 | D0 | 1      | K0   | K1   | A1  | B1  | NaN | NaN |
| 1    | K0   | K1   | A1 | B1 | 1     | K1   | K0   | C1 | D1 | 2      | K1   | K0   | A2  | B2  | C1  | D1  |
| 2    | K1   | K0   | A2 | B2 | 2     | K1   | K0   | C2 | D2 | 3      | K1   | K0   | A2  | B2  | C2  | D2  |
| 3    | K2   | K1   | A3 | B3 | 3     | K2   | K0   | C3 | D3 | 4      | K2   | K1   | A3  | B3  | NaN | NaN |
|      |      |      |    |    |       |      |      |    |    | 5      | K2   | K0   | NaN | NaN | C3  | D3  |

```
In [47]: result = pd.merge(left, right, how='inner', on=['key1', 'key2'])
```

| left |      |      |    |    | right |      |      |    |    | Result |      |      |    |    |    |    |
|------|------|------|----|----|-------|------|------|----|----|--------|------|------|----|----|----|----|
|      | key1 | key2 | A  | B  |       | key1 | key2 | C  | D  |        | key1 | key2 | A  | B  | C  | D  |
| 0    | K0   | K0   | A0 | B0 | 0     | K0   | K0   | C0 | D0 | 0      | K0   | K0   | A0 | B0 | C0 | D0 |
| 1    | K0   | K1   | A1 | B1 | 1     | K1   | K0   | C1 | D1 | 1      | K1   | K0   | A2 | B2 | C1 | D1 |
| 2    | K1   | K0   | A2 | B2 | 2     | K1   | K0   | C2 | D2 | 2      | K1   | K0   | A2 | B2 | C2 | D2 |
| 3    | K2   | K1   | A3 | B3 | 3     | K2   | K0   | C3 | D3 |        |      |      |    |    |    |    |

Here is another example with duplicate join keys in DataFrames:

```
In [48]: left = pd.DataFrame({'A' : [1,2], 'B' : [2, 2]})
```

```
In [49]: right = pd.DataFrame({'A' : [4,5,6], 'B': [2,2,2]})
```

```
In [50]: result = pd.merge(left, right, on='B', how='outer')
```

| left |   |   | right |   |   | Result |     |   |     |
|------|---|---|-------|---|---|--------|-----|---|-----|
|      | A | B |       | A | B |        | A_x | B | A_y |
| 0    | 1 | 2 | 0     | 4 | 2 | 0      | 1   | 2 | 4   |
| 1    | 2 | 2 | 1     | 5 | 2 | 1      | 1   | 2 | 5   |
|      |   |   | 2     | 6 | 2 | 2      | 1   | 2 | 6   |
|      |   |   |       |   |   | 3      | 2   | 2 | 4   |
|      |   |   |       |   |   | 4      | 2   | 2 | 5   |
|      |   |   |       |   |   | 5      | 2   | 2 | 6   |

**Warning:** Joining / merging on duplicate keys can cause a returned frame that is the multiplication of the row dimensions, which may result in memory overflow. It is the user's responsibility to manage duplicate values in keys before joining large DataFrames.

## 17.2.2 Checking for duplicate keys

New in version 0.21.0.

Users can use the `validate` argument to automatically check whether there are unexpected duplicates in their merge keys. Key uniqueness is checked before merge operations and so should protect against memory overflows. Checking key uniqueness is also a good way to ensure user data structures are as expected.

In the following example, there are duplicate values of `B` in the right DataFrame. As this is not a one-to-one merge – as specified in the `validate` argument – an exception will be raised.

```
In [51]: left = pd.DataFrame({'A' : [1,2], 'B' : [1, 2]})
In [52]: right = pd.DataFrame({'A' : [4,5,6], 'B': [2, 2, 2]})
```

```
In [53]: result = pd.merge(left, right, on='B', how='outer', validate="one_to_one")
...
MergeError: Merge keys are not unique in right dataset; not a one-to-one merge
```

If the user is aware of the duplicates in the right DataFrame but wants to ensure there are no duplicates in the left DataFrame, one can use the `validate='one_to_many'` argument instead, which will not raise an exception.

```
In [53]: pd.merge(left, right, on='B', how='outer', validate="one_to_many")
Out[53]:
```

|   | A_x | B | A_y |
|---|-----|---|-----|
| 0 | 1   | 1 | NaN |
| 1 | 2   | 2 | 4.0 |
| 2 | 2   | 2 | 5.0 |
| 3 | 2   | 2 | 6.0 |

## 17.2.3 The merge indicator

`merge()` accepts the argument `indicator`. If `True`, a Categorical-type column called `_merge` will be added to the output object that takes on values:

| Observation Origin              | _merge value |
|---------------------------------|--------------|
| Merge key only in 'left' frame  | left_only    |
| Merge key only in 'right' frame | right_only   |
| Merge key in both frames        | both         |

```
In [54]: df1 = pd.DataFrame({'coll': [0, 1], 'col_left':['a', 'b']})
In [55]: df2 = pd.DataFrame({'coll': [1, 2, 2], 'col_right':[2, 2, 2]})
In [56]: pd.merge(df1, df2, on='coll', how='outer', indicator=True)
Out[56]:
```

|   | coll | col_left | col_right | _merge     |
|---|------|----------|-----------|------------|
| 0 | 0    | a        | NaN       | left_only  |
| 1 | 1    | b        | 2.0       | both       |
| 2 | 2    | NaN      | 2.0       | right_only |
| 3 | 2    | NaN      | 2.0       | right_only |

The `indicator` argument will also accept string arguments, in which case the indicator function will use the value of the passed string as the name for the indicator column.

```
In [57]: pd.merge(df1, df2, on='coll', how='outer', indicator='indicator_column')
Out[57]:
```

|   | coll | col_left | col_right | indicator_column |
|---|------|----------|-----------|------------------|
| 0 | 0    | a        | NaN       | left_only        |
| 1 | 1    | b        | 2.0       | both             |
| 2 | 2    | NaN      | 2.0       | right_only       |
| 3 | 2    | NaN      | 2.0       | right_only       |

## 17.2.4 Merge Dtypes

New in version 0.19.0.

Merging will preserve the dtype of the join keys.

```
In [58]: left = pd.DataFrame({'key': [1], 'v1': [10]})
In [59]: left
Out[59]:
```

|   | key | v1 |
|---|-----|----|
| 0 | 1   | 10 |

```
In [60]: right = pd.DataFrame({'key': [1, 2], 'v1': [20, 30]})
In [61]: right
Out[61]:
```

|   | key | v1 |
|---|-----|----|
| 0 | 1   | 20 |
| 1 | 2   | 30 |

We are able to preserve the join keys:

```
In [62]: pd.merge(left, right, how='outer')
Out[62]:
```

|  | key | v1 |
|--|-----|----|
|--|-----|----|

(continues on next page)

(continued from previous page)

```

0    1    10
1    1    20
2    2    30

In [63]: pd.merge(left, right, how='outer').dtypes
Out[63]:
key      int64
v1       int64
dtype: object

```

Of course if you have missing values that are introduced, then the resulting dtype will be upcast.

```

In [64]: pd.merge(left, right, how='outer', on='key')
Out[64]:
   key  v1_x  v1_y
0    1  10.0    20
1    2   NaN    30

In [65]: pd.merge(left, right, how='outer', on='key').dtypes
Out[65]:
key      int64
v1_x    float64
v1_y     int64
dtype: object

```

New in version 0.20.0.

Merging will preserve category dtypes of the mergands. See also the section on *categoricals*.

The left frame.

```

In [66]: from pandas.api.types import CategoricalDtype

In [67]: X = pd.Series(np.random.choice(['foo', 'bar'], size=(10,)))

In [68]: X = X.astype(CategoricalDtype(categories=['foo', 'bar']))

In [69]: left = pd.DataFrame({'X': X,
.....:                       'Y': np.random.choice(['one', 'two', 'three'], size=(10,
.....:                                                ↪))})
.....:

In [70]: left
Out[70]:
   X      Y
0  bar  one
1  foo  one
2  foo three
3  bar three
4  foo  one
5  bar  one
6  bar three
7  bar three
8  bar three
9  foo three

In [71]: left.dtypes

```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
X      category
Y      object
dtype: object

```

The right frame.

```

In [72]: right = pd.DataFrame({
.....:     'X': pd.Series(['foo', 'bar'],
.....:                     dtype=CategoricalDtype(['foo', 'bar'])),
.....:     'Z': [1, 2]
.....: })
.....:

In [73]: right
Out[73]:
   X  Z
0  foo  1
1  bar  2

In [74]: right.dtypes
Out[74]:
X      category
Z      int64
dtype: object

```

The merged result:

```

In [75]: result = pd.merge(left, right, how='outer')

In [76]: result
Out[76]:
   X      Y  Z
0  bar  one  2
1  bar three  2
2  bar  one  2
3  bar three  2
4  bar three  2
5  bar three  2
6  foo  one  1
7  foo three  1
8  foo  one  1
9  foo three  1

In [77]: result.dtypes
Out[77]:
↪
X      category
Y      object
Z      int64
dtype: object

```

**Note:** The category dtypes must be *exactly* the same, meaning the same categories and the ordered attribute. Otherwise the result will coerce to object dtype.

**Note:** Merging on category dtypes that are the same can be quite performant compared to object dtype merging.

## 17.2.5 Joining on index

`DataFrame.join()` is a convenient method for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame. Here is a very basic example:

```
In [78]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
.....:                       'B': ['B0', 'B1', 'B2']},
.....:                       index=['K0', 'K1', 'K2'])

In [79]: right = pd.DataFrame({'C': ['C0', 'C2', 'C3'],
.....:                         'D': ['D0', 'D2', 'D3']},
.....:                         index=['K0', 'K2', 'K3'])

In [80]: result = left.join(right)
```

| left |    |    | right |    |    | Result |    |    |     |     |
|------|----|----|-------|----|----|--------|----|----|-----|-----|
|      | A  | B  |       | C  | D  |        | A  | B  | C   | D   |
| K0   | A0 | B0 | K0    | C0 | D0 | K0     | A0 | B0 | C0  | D0  |
| K1   | A1 | B1 | K2    | C2 | D2 | K1     | A1 | B1 | NaN | NaN |
| K2   | A2 | B2 | K3    | C3 | D3 | K2     | A2 | B2 | C2  | D2  |

```
In [81]: result = left.join(right, how='outer')
```

| left |    |    | right |    |    | Result |     |     |     |     |
|------|----|----|-------|----|----|--------|-----|-----|-----|-----|
|      | A  | B  |       | C  | D  |        | A   | B   | C   | D   |
| K0   | A0 | B0 | K0    | C0 | D0 | K0     | A0  | B0  | C0  | D0  |
| K1   | A1 | B1 | K2    | C2 | D2 | K1     | A1  | B1  | NaN | NaN |
| K2   | A2 | B2 | K3    | C3 | D3 | K2     | A2  | B2  | C2  | D2  |
|      |    |    |       |    |    | K3     | NaN | NaN | C3  | D3  |

The same as above, but with `how='inner'`.

```
In [82]: result = left.join(right, how='inner')
```

| left |    |    | right |    |    | Result |    |    |    |    |
|------|----|----|-------|----|----|--------|----|----|----|----|
|      | A  | B  |       | C  | D  |        | A  | B  | C  | D  |
| K0   | A0 | B0 | K0    | C0 | D0 | K0     | A0 | B0 | C0 | D0 |
| K1   | A1 | B1 | K2    | C2 | D2 |        |    |    |    |    |
| K2   | A2 | B2 | K3    | C3 | D3 | K2     | A2 | B2 | C2 | D2 |



The data alignment here is on the indexes (row labels). This same behavior can be achieved using `merge` plus additional arguments instructing it to use the indexes:

```
In [83]: result = pd.merge(left, right, left_index=True, right_index=True, how='outer')
↳')
```

| left |    |    | right |    |    | Result |     |     |     |     |
|------|----|----|-------|----|----|--------|-----|-----|-----|-----|
|      | A  | B  |       | C  | D  |        | A   | B   | C   | D   |
| K0   | A0 | B0 | K0    | C0 | D0 | K0     | A0  | B0  | C0  | D0  |
| K1   | A1 | B1 | K2    | C2 | D2 | K1     | A1  | B1  | NaN | NaN |
| K2   | A2 | B2 | K3    | C3 | D3 | K2     | A2  | B2  | C2  | D2  |
|      |    |    |       |    |    | K3     | NaN | NaN | C3  | D3  |

```
In [84]: result = pd.merge(left, right, left_index=True, right_index=True, how='inner')
↳');
```

| left |    |    | right |    |    | Result |    |    |    |    |
|------|----|----|-------|----|----|--------|----|----|----|----|
|      | A  | B  |       | C  | D  |        | A  | B  | C  | D  |
| K0   | A0 | B0 | K0    | C0 | D0 | K0     | A0 | B0 | C0 | D0 |
| K1   | A1 | B1 | K2    | C2 | D2 | K2     | A2 | B2 | C2 | D2 |
| K2   | A2 | B2 | K3    | C3 | D3 |        |    |    |    |    |

## 17.2.6 Joining key columns on an index

`join()` takes an optional `on` argument which may be a column or multiple column names, which specifies that the passed `DataFrame` is to be aligned on that column in the `DataFrame`. These two function calls are completely equivalent:

```
left.join(right, on=key_or_keys)
pd.merge(left, right, left_on=key_or_keys, right_index=True,
        how='left', sort=False)
```

Obviously you can choose whichever form you find more convenient. For many-to-one joins (where one of the `DataFrame`'s is already indexed by the join key), using `join` may be more convenient. Here is a simple example:

```
In [85]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
.....:                      'B': ['B0', 'B1', 'B2', 'B3'],
.....:                      'key': ['K0', 'K1', 'K0', 'K1']})
.....:

In [86]: right = pd.DataFrame({'C': ['C0', 'C1'],
.....:                       'D': ['D0', 'D1']},
.....:                       index=['K0', 'K1'])
.....:

In [87]: result = left.join(right, on='key')
```

| left |    |    |     | right |    |    | Result |    |    |     |    |    |
|------|----|----|-----|-------|----|----|--------|----|----|-----|----|----|
|      | A  | B  | key |       | C  | D  |        | A  | B  | key | C  | D  |
| 0    | A0 | B0 | K0  |       |    |    | 0      | A0 | B0 | K0  | C0 | D0 |
| 1    | A1 | B1 | K1  | K0    | C0 | D0 | 1      | A1 | B1 | K1  | C1 | D1 |
| 2    | A2 | B2 | K0  | K1    | C1 | D1 | 2      | A2 | B2 | K0  | C0 | D0 |
| 3    | A3 | B3 | K1  |       |    |    | 3      | A3 | B3 | K1  | C1 | D1 |

```
In [88]: result = pd.merge(left, right, left_on='key', right_index=True,
.....:                     how='left', sort=False);
.....:
```

| left |    |    |     | right |    |    | Result |    |    |     |    |    |
|------|----|----|-----|-------|----|----|--------|----|----|-----|----|----|
|      | A  | B  | key |       | C  | D  |        | A  | B  | key | C  | D  |
| 0    | A0 | B0 | K0  |       |    |    | 0      | A0 | B0 | K0  | C0 | D0 |
| 1    | A1 | B1 | K1  | K0    | C0 | D0 | 1      | A1 | B1 | K1  | C1 | D1 |
| 2    | A2 | B2 | K0  | K1    | C1 | D1 | 2      | A2 | B2 | K0  | C0 | D0 |
| 3    | A3 | B3 | K1  |       |    |    | 3      | A3 | B3 | K1  | C1 | D1 |

To join on multiple keys, the

passed DataFrame must have a MultiIndex:

```
In [89]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
.....:                       'B': ['B0', 'B1', 'B2', 'B3'],
.....:                       'key1': ['K0', 'K0', 'K1', 'K2'],
.....:                       'key2': ['K0', 'K1', 'K0', 'K1']})

In [90]: index = pd.MultiIndex.from_tuples([('K0', 'K0'), ('K1', 'K0'),
.....:                                     ('K2', 'K0'), ('K2', 'K1')])

In [91]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
.....:                         'D': ['D0', 'D1', 'D2', 'D3']},
.....:                        index=index)
.....:
```

Now this can be joined by passing the two key column names:

```
In [92]: result = left.join(right, on=['key1', 'key2'])
```

| left |    |    |      |      | right |    |    |    | Result |    |    |      |      |     |     |
|------|----|----|------|------|-------|----|----|----|--------|----|----|------|------|-----|-----|
|      | A  | B  | key1 | key2 |       |    | C  | D  |        | A  | B  | key1 | key2 | C   | D   |
| 0    | A0 | B0 | K0   | K0   | K0    | K0 | C0 | D0 | 0      | A0 | B0 | K0   | K0   | C0  | D0  |
| 1    | A1 | B1 | K0   | K1   | K1    | K0 | C1 | D1 | 1      | A1 | B1 | K0   | K1   | NaN | NaN |
| 2    | A2 | B2 | K1   | K0   | K2    | K0 | C2 | D2 | 2      | A2 | B2 | K1   | K0   | C1  | D1  |
| 3    | A3 | B3 | K2   | K1   | K2    | K1 | C3 | D3 | 3      | A3 | B3 | K2   | K1   | C3  | D3  |

The default for `DataFrame.join` is to perform a left join (essentially a “VLOOKUP” operation, for Excel users), which

uses only the keys found in the calling DataFrame. Other join types, for example inner join, can be just as easily performed:

```
In [93]: result = left.join(right, on=['key1', 'key2'], how='inner')
```

| left |    |    |      |      | right |    |    |    | Result |    |    |      |      |    |    |
|------|----|----|------|------|-------|----|----|----|--------|----|----|------|------|----|----|
|      | A  | B  | key1 | key2 |       |    | C  | D  |        | A  | B  | key1 | key2 | C  | D  |
| 0    | A0 | B0 | K0   | K0   | K0    | K0 | C0 | D0 | 0      | A0 | B0 | K0   | K0   | C0 | D0 |
| 1    | A1 | B1 | K0   | K1   | K1    | K0 | C1 | D1 | 2      | A2 | B2 | K1   | K0   | C1 | D1 |
| 2    | A2 | B2 | K1   | K0   | K2    | K0 | C2 | D2 | 3      | A3 | B3 | K2   | K1   | C3 | D3 |
| 3    | A3 | B3 | K2   | K1   | K2    | K1 | C3 | D3 |        |    |    |      |      |    |    |

As you can see, this drops any rows where there was no match.

### 17.2.7 Joining a single Index to a Multi-index

You can join a singly-indexed DataFrame with a level of a multi-indexed DataFrame. The level will match on the name of the index of the singly-indexed frame against a level name of the multi-indexed frame.

```
In [94]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
.....:                       'B': ['B0', 'B1', 'B2']},
.....:                       index=pd.Index(['K0', 'K1', 'K2'], name='key'))
.....:
```

```
In [95]: index = pd.MultiIndex.from_tuples([('K0', 'Y0'), ('K1', 'Y1'),
.....:                                     ('K2', 'Y2'), ('K2', 'Y3')],
.....:                                     names=['key', 'Y'])
.....:
```

```
In [96]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
.....:                        'D': ['D0', 'D1', 'D2', 'D3']},
.....:                        index=index)
.....:
```

```
In [97]: result = left.join(right, how='inner')
```

| left |    |    | right |    |    |    | Result |    |    |    |    |    |
|------|----|----|-------|----|----|----|--------|----|----|----|----|----|
|      | A  | B  |       |    | C  | D  |        |    | A  | B  | C  | D  |
| K0   | A0 | B0 | K0    | Y0 | C0 | D0 | K0     | Y0 | A0 | B0 | C0 | D0 |
| K1   | A1 | B1 | K1    | Y1 | C1 | D1 | K1     | Y1 | A1 | B1 | C1 | D1 |
| K2   | A2 | B2 | K2    | Y2 | C2 | D2 | K2     | Y2 | A2 | B2 | C2 | D2 |
|      |    |    | K2    | Y3 | C3 | D3 | K2     | Y3 | A2 | B2 | C3 | D3 |

This is equivalent but less verbose and more memory efficient / faster than this.

```
In [98]: result = pd.merge(left.reset_index(), right.reset_index(),
.....:                      on=['key', 'Y'], how='inner').set_index(['key', 'Y'])
.....:
```

| left |    |    | right |    |    |    | Result |    |    |    |    |    |  |  |
|------|----|----|-------|----|----|----|--------|----|----|----|----|----|--|--|
|      |    |    |       |    | C  | D  |        |    | A  | B  | C  | D  |  |  |
| K0   | A0 | B0 | K0    | Y0 | C0 | D0 | K0     | Y0 | A0 | B0 | C0 | D0 |  |  |
| K1   | A1 | B1 | K1    | Y1 | C1 | D1 | K1     | Y1 | A1 | B1 | C1 | D1 |  |  |
| K2   | A2 | B2 | K2    | Y2 | C2 | D2 | K2     | Y2 | A2 | B2 | C2 | D2 |  |  |
|      |    |    | K2    | Y3 | C3 | D3 | K2     | Y3 | A2 | B2 | C3 | D3 |  |  |

## 17.2.8 Joining with two multi-indexes

This is not implemented via `join` at-the-moment, however it can be done using the following code.

```
In [99]: index = pd.MultiIndex.from_tuples([('K0', 'X0'), ('K0', 'X1'),
.....:                                     ('K1', 'X2')],
.....:                                     names=['key', 'X'])
.....:

In [100]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2'],
.....:                         'B': ['B0', 'B1', 'B2']},
.....:                        index=index)
.....:

In [101]: result = pd.merge(left.reset_index(), right.reset_index(),
.....:                       on=['key'], how='inner').set_index(['key', 'X', 'Y'])
.....:
```

| left |    |    |    | right |    |    |    | Result |    |    |    |    |    |    |  |
|------|----|----|----|-------|----|----|----|--------|----|----|----|----|----|----|--|
|      |    | A  | B  |       |    | C  | D  |        |    |    | A  | B  | C  | D  |  |
| K0   | X0 | A0 | B0 | K0    | Y0 | C0 | D0 | K0     | X0 | Y0 | A0 | B0 | C0 | D0 |  |
| K0   | X1 | A1 | B1 | K1    | Y1 | C1 | D1 | K0     | X1 | Y0 | A1 | B1 | C0 | D0 |  |
| K1   | X2 | A2 | B2 | K2    | Y2 | C2 | D2 | K1     | X2 | Y1 | A2 | B2 | C1 | D1 |  |
|      |    |    |    | K2    | Y3 | C3 | D3 |        |    |    |    |    |    |    |  |

## 17.2.9 Merging on a combination of columns and index levels

New in version 0.22.

Strings passed as the `on`, `left_on`, and `right_on` parameters may refer to either column names or index level names. This enables merging `DataFrame` instances on a combination of index levels and columns without resetting indexes.

```
In [102]: left_index = pd.Index(['K0', 'K0', 'K1', 'K2'], name='key1')

In [103]: left = pd.DataFrame({'A': ['A0', 'A1', 'A2', 'A3'],
.....:                         'B': ['B0', 'B1', 'B2', 'B3'],
.....:                         'key2': ['K0', 'K1', 'K0', 'K1']},
.....:                        index=left_index)
```

(continues on next page)

(continued from previous page)

```

.....:
In [104]: right_index = pd.Index(['K0', 'K1', 'K2', 'K2'], name='key1')

In [105]: right = pd.DataFrame({'C': ['C0', 'C1', 'C2', 'C3'],
.....:                        'D': ['D0', 'D1', 'D2', 'D3'],
.....:                        'key2': ['K0', 'K0', 'K0', 'K1']},
.....:                        index=right_index)
.....:

In [106]: result = left.merge(right, on=['key1', 'key2'])

```

| left |    |    |      | right |    |    |      | Result |    |    |      |    |    |
|------|----|----|------|-------|----|----|------|--------|----|----|------|----|----|
|      | A  | B  | key2 |       | C  | D  | key2 |        | A  | B  | key2 | C  | D  |
| K0   | A0 | B0 | K0   | K0    | C0 | D0 | K0   | K0     | A0 | B0 | K0   | C0 | D0 |
| K0   | A1 | B1 | K1   | K1    | C1 | D1 | K0   | K1     | A2 | B2 | K0   | C1 | D1 |
| K1   | A2 | B2 | K0   | K2    | C2 | D2 | K0   | K2     | A3 | B3 | K1   | C3 | D3 |
| K2   | A3 | B3 | K1   | K2    | C3 | D3 | K1   |        |    |    |      |    |    |

**Note:** When DataFrames are merged on a string that matches an index level in both frames, the index level is preserved as an index level in the resulting DataFrame.

**Note:** If a string matches both a column name and an index level name, then a warning is issued and the column takes precedence. This will result in an ambiguity error in a future version.

### 17.2.10 Overlapping value columns

The merge `suffixes` argument takes a tuple of list of strings to append to overlapping column names in the input DataFrames to disambiguate the result columns:

```

In [107]: left = pd.DataFrame({'k': ['K0', 'K1', 'K2'], 'v': [1, 2, 3]})

In [108]: right = pd.DataFrame({'k': ['K0', 'K0', 'K3'], 'v': [4, 5, 6]})

In [109]: result = pd.merge(left, right, on='k')

```

| left |    |   | right |    |   | Result |    |     |     |
|------|----|---|-------|----|---|--------|----|-----|-----|
|      | k  | v |       | k  | v |        | k  | v_x | v_y |
| 0    | K0 | 1 | 0     | K0 | 4 | 0      | K0 | 1   | 4   |
| 1    | K1 | 2 | 1     | K0 | 5 | 1      | K0 | 1   | 5   |
| 2    | K2 | 3 | 2     | K3 | 6 |        |    |     |     |

```
In [110]: result = pd.merge(left, right, on='k', suffixes=['_l', '_r'])
```

| left |    |   | right |    |   | Result |    |     |     |
|------|----|---|-------|----|---|--------|----|-----|-----|
|      | k  | v |       | k  | v |        | k  | v_l | v_r |
| 0    | K0 | 1 | 0     | K0 | 4 | 0      | K0 | 1   | 4   |
| 1    | K1 | 2 | 1     | K0 | 5 | 1      | K0 | 1   | 5   |
| 2    | K2 | 3 | 2     | K3 | 6 |        |    |     |     |

`DataFrame.join()` has `lsuffix` and `rsuffix` arguments which behave similarly.

```
In [111]: left = left.set_index('k')
```

```
In [112]: right = right.set_index('k')
```

```
In [113]: result = left.join(right, lsuffix='_l', rsuffix='_r')
```

| left |   | right |   | Result |     |     |
|------|---|-------|---|--------|-----|-----|
|      | v |       | v |        | v_l | v_r |
| K0   | 1 | K0    | 4 | K0     | 1   | 4.0 |
| K1   | 2 | K0    | 5 | K0     | 1   | 5.0 |
| K2   | 3 | K3    | 6 | K1     | 2   | NaN |
|      |   |       |   | K2     | 3   | NaN |

### 17.2.11 Joining multiple DataFrame or Panel objects

A list or tuple of DataFrames can also be passed to `join()` to join them together on their indexes.

```
In [114]: right2 = pd.DataFrame({'v': [7, 8, 9]}, index=['K1', 'K1', 'K2'])
```

```
In [115]: result = left.join([right, right2])
```

| left |   | right |   | right2 |   | Result |     |     |     |
|------|---|-------|---|--------|---|--------|-----|-----|-----|
|      | v |       | v |        | v |        | v_x | v_y | v   |
| K0   | 1 | K0    | 4 | K1     | 7 | K0     | 1   | 4.0 | NaN |
| K1   | 2 | K0    | 5 | K1     | 8 | K0     | 1   | 5.0 | NaN |
| K2   | 3 | K3    | 6 | K2     | 9 | K1     | 2   | NaN | 7.0 |
|      |   |       |   |        |   | K1     | 2   | NaN | 8.0 |
|      |   |       |   |        |   | K2     | 3   | NaN | 9.0 |

## 17.2.12 Merging together values within Series or DataFrame columns

Another fairly common situation is to have two like-indexed (or similarly indexed) `Series` or `DataFrame` objects and wanting to “patch” values in one object from values for matching indices in the other. Here is an example:

```
In [116]: df1 = pd.DataFrame([[np.nan, 3., 5.], [-4.6, np.nan, np.nan],
.....:                        [np.nan, 7., np.nan]])
.....:

In [117]: df2 = pd.DataFrame([[ -42.6, np.nan, -8.2], [-5., 1.6, 4]],
.....:                        index=[1, 2])
.....:
```

For this, use the `combine_first()` method:

```
In [118]: result = df1.combine_first(df2)
```

| df1 |      |     |     | df2 |       |     |      | Result |      |     |      |
|-----|------|-----|-----|-----|-------|-----|------|--------|------|-----|------|
|     | 0    | 1   | 2   |     | 0     | 1   | 2    |        | 0    | 1   | 2    |
| 0   | NaN  | 3.0 | 5.0 |     |       |     |      | 0      | NaN  | 3.0 | 5.0  |
| 1   | -4.6 | NaN | NaN | 1   | -42.6 | NaN | -8.2 | 1      | -4.6 | NaN | -8.2 |
| 2   | NaN  | 7.0 | NaN | 2   | -5.0  | 1.6 | 4.0  | 2      | -5.0 | 7.0 | 4.0  |

Note that this method only takes values from the right `DataFrame` if they are missing in the left `DataFrame`. A related method, `update()`, alters non-NA values inplace:

```
In [119]: df1.update(df2)
```

| df1 |      |     |     | df2 |       |     |      | Result |       |     |      |
|-----|------|-----|-----|-----|-------|-----|------|--------|-------|-----|------|
|     | 0    | 1   | 2   |     | 0     | 1   | 2    |        | 0     | 1   | 2    |
| 0   | NaN  | 3.0 | 5.0 |     |       |     |      | 0      | NaN   | 3.0 | 5.0  |
| 1   | -4.6 | NaN | NaN | 1   | -42.6 | NaN | -8.2 | 1      | -42.6 | NaN | -8.2 |
| 2   | NaN  | 7.0 | NaN | 2   | -5.0  | 1.6 | 4.0  | 2      | -5.0  | 1.6 | 4.0  |

## 17.3 Timeseries friendly merging

### 17.3.1 Merging Ordered Data

A `merge_ordered()` function allows combining time series and other ordered data. In particular it has an optional `fill_method` keyword to fill/interpolate missing data:

```
In [120]: left = pd.DataFrame({'k': ['K0', 'K1', 'K1', 'K2'],
.....:                        'lv': [1, 2, 3, 4],
.....:                        's': ['a', 'b', 'c', 'd']})
.....:

In [121]: right = pd.DataFrame({'k': ['K1', 'K2', 'K4'],
```

(continues on next page)

(continued from previous page)

```

.....:             'rv': [1, 2, 3])
.....:

In [122]: pd.merge_ordered(left, right, fill_method='ffill', left_by='s')
Out[122]:
   k  lv s  rv
0  K0  1.0 a NaN
1  K1  1.0 a  1.0
2  K2  1.0 a  2.0
3  K4  1.0 a  3.0
4  K1  2.0 b  1.0
5  K2  2.0 b  2.0
6  K4  2.0 b  3.0
7  K1  3.0 c  1.0
8  K2  3.0 c  2.0
9  K4  3.0 c  3.0
10 K1  NaN d  1.0
11 K2  4.0 d  2.0
12 K4  4.0 d  3.0

```

## 17.3.2 Merging AsOf

New in version 0.19.0.

A `merge_asof()` is similar to an ordered left-join except that we match on nearest key rather than equal keys. For each row in the left DataFrame, we select the last row in the right DataFrame whose on key is less than the left's key. Both DataFrames must be sorted by the key.

Optionally an asof merge can perform a group-wise merge. This matches the `by` key equally, in addition to the nearest match on the `on` key.

For example; we might have `trades` and `quotes` and we want to asof merge them.

```

In [123]: trades = pd.DataFrame({
.....:     'time': pd.to_datetime(['20160525 13:30:00.023',
.....:                             '20160525 13:30:00.038',
.....:                             '20160525 13:30:00.048',
.....:                             '20160525 13:30:00.048',
.....:                             '20160525 13:30:00.048']),
.....:     'ticker': ['MSFT', 'MSFT',
.....:                 'GOOG', 'GOOG', 'AAPL'],
.....:     'price': [51.95, 51.95,
.....:               720.77, 720.92, 98.00],
.....:     'quantity': [75, 155,
.....:                  100, 100, 100]},
.....:     columns=['time', 'ticker', 'price', 'quantity'])
.....:

In [124]: quotes = pd.DataFrame({
.....:     'time': pd.to_datetime(['20160525 13:30:00.023',
.....:                             '20160525 13:30:00.023',
.....:                             '20160525 13:30:00.030',
.....:                             '20160525 13:30:00.041',
.....:                             '20160525 13:30:00.048',
.....:                             '20160525 13:30:00.049',
.....:                             '20160525 13:30:00.072'],
.....:

```

(continues on next page)



By default we are taking the asof of the quotes.

We only asof within 2ms between the quote time and the trade time.

(continues on next page)

(continued from previous page)

|   |                         |      |        |     |        |        |
|---|-------------------------|------|--------|-----|--------|--------|
| 0 | 2016-05-25 13:30:00.023 | MSFT | 51.95  | 75  | 51.95  | 51.96  |
| 1 | 2016-05-25 13:30:00.038 | MSFT | 51.95  | 155 | NaN    | NaN    |
| 2 | 2016-05-25 13:30:00.048 | GOOG | 720.77 | 100 | 720.50 | 720.93 |
| 3 | 2016-05-25 13:30:00.048 | GOOG | 720.92 | 100 | 720.50 | 720.93 |
| 4 | 2016-05-25 13:30:00.048 | AAPL | 98.00  | 100 | NaN    | NaN    |

We only asof within 10ms between the quote time and the trade time and we exclude exact matches on time. Note that though we exclude the exact matches (of the quotes), prior quotes **do** propagate to that point in time.

```
In [129]: pd.merge_asof(trades, quotes,
.....:                  on='time',
.....:                  by='ticker',
.....:                  tolerance=pd.Timedelta('10ms'),
.....:                  allow_exact_matches=False)
.....:
```

```
Out [129]:
```

|   | time                    | ticker | price  | quantity | bid   | ask   |
|---|-------------------------|--------|--------|----------|-------|-------|
| 0 | 2016-05-25 13:30:00.023 | MSFT   | 51.95  | 75       | NaN   | NaN   |
| 1 | 2016-05-25 13:30:00.038 | MSFT   | 51.95  | 155      | 51.97 | 51.98 |
| 2 | 2016-05-25 13:30:00.048 | GOOG   | 720.77 | 100      | NaN   | NaN   |
| 3 | 2016-05-25 13:30:00.048 | GOOG   | 720.92 | 100      | NaN   | NaN   |
| 4 | 2016-05-25 13:30:00.048 | AAPL   | 98.00  | 100      | NaN   | NaN   |

## RESHAPING AND PIVOT TABLES

### 18.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```
In [1]: df
Out[1]:
```

|    | date       | variable | value     |
|----|------------|----------|-----------|
| 0  | 2000-01-03 | A        | 0.469112  |
| 1  | 2000-01-04 | A        | -0.282863 |
| 2  | 2000-01-05 | A        | -1.509059 |
| 3  | 2000-01-03 | B        | -1.135632 |
| 4  | 2000-01-04 | B        | 1.212112  |
| 5  | 2000-01-05 | B        | -0.173215 |
| 6  | 2000-01-03 | C        | 0.119209  |
| 7  | 2000-01-04 | C        | -1.044236 |
| 8  | 2000-01-05 | C        | -0.861849 |
| 9  | 2000-01-03 | D        | -2.104569 |
| 10 | 2000-01-04 | D        | -0.494929 |
| 11 | 2000-01-05 | D        | 1.071804  |

For the curious here is how the above DataFrame was created:

```
import pandas.util.testing as tm; tm.N = 3
def unpivot(frame):
    N, K = frame.shape
    data = {'value' : frame.values.ravel('F'),
           'variable' : np.asarray(frame.columns).repeat(N),
           'date' : np.tile(np.asarray(frame.index), K)}
    return pd.DataFrame(data, columns=['date', 'variable', 'value'])
df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable A we could do:

```
In [2]: df[df['variable'] == 'A']
Out[2]:
```

|   | date       | variable | value     |
|---|------------|----------|-----------|
| 0 | 2000-01-03 | A        | 0.469112  |
| 1 | 2000-01-04 | A        | -0.282863 |
| 2 | 2000-01-05 | A        | -1.509059 |

# Pivot

df

|   | foo | bar | baz | zoo |
|---|-----|-----|-----|-----|
| 0 | one | A   | 1   | x   |
| 1 | one | B   | 2   | y   |
| 2 | one | C   | 3   | z   |
| 3 | two | A   | 4   | q   |
| 4 | two | B   | 5   | w   |
| 5 | two | C   | 6   | t   |

➔

```
df.pivot(index='foo',
          columns='bar',
          values='baz')
```

|     | bar | A | B | C |
|-----|-----|---|---|---|
| foo |     |   |   |   |
| one |     | 1 | 2 | 3 |
| two |     | 4 | 5 | 6 |

But suppose we wish to do time series operations with the variables. A better representation would be where the columns are the unique variables and an index of dates identifies individual observations. To reshape the data into this form, we use the `DataFrame.pivot()` method (also implemented as a top level function `pivot()`):

```
In [3]: df.pivot(index='date', columns='variable', values='value')
Out[3]:
```

| variable   | A         | B         | C         | D         |
|------------|-----------|-----------|-----------|-----------|
| date       |           |           |           |           |
| 2000-01-03 | 0.469112  | -1.135632 | 0.119209  | -2.104569 |
| 2000-01-04 | -0.282863 | 1.212112  | -1.044236 | -0.494929 |
| 2000-01-05 | -1.509059 | -0.173215 | -0.861849 | 1.071804  |

If the `values` argument is omitted, and the input `DataFrame` has more than one column of values which are not used as column or index inputs to `pivot`, then the resulting “pivoted” `DataFrame` will have *hierarchical columns* whose topmost level indicates the respective value column:

```
In [4]: df['value2'] = df['value'] * 2
In [5]: pivoted = df.pivot('date', 'variable')
In [6]: pivoted
Out[6]:
```

|            | value     |           |           |           | value2    |           |           |           |
|------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| variable   | A         | B         | C         | D         | A         | B         | C         | D         |
| date       |           |           |           |           |           |           |           |           |
| 2000-01-03 | 0.469112  | -1.135632 | 0.119209  | -2.104569 | 0.938225  | -2.271265 | 0.238417  | -4.209138 |
| 2000-01-04 | -0.282863 | 1.212112  | -1.044236 | -0.494929 | -0.565727 | 2.424224  | -2.088472 | -0.989859 |
| 2000-01-05 | -1.509059 | -0.173215 | -0.861849 | 1.071804  | -3.018117 | -0.346429 | -1.723698 | 2.143608  |

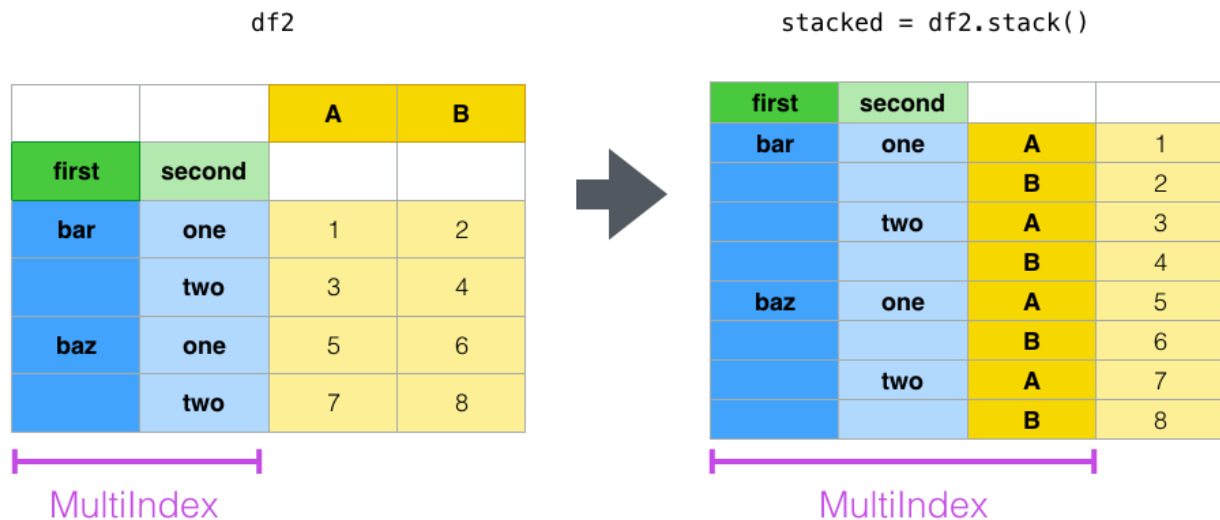
You can then select subsets from the pivoted `DataFrame`:

```
In [7]: pivoted['value2']
Out [7]:
variable          A          B          C          D
date
2000-01-03  0.938225 -2.271265  0.238417 -4.209138
2000-01-04 -0.565727  2.424224 -2.088472 -0.989859
2000-01-05 -3.018117 -0.346429 -1.723698  2.143608
```

Note that this returns a view on the underlying data in the case where the data are homogeneously-typed.

## 18.2 Reshaping by stacking and unstacking

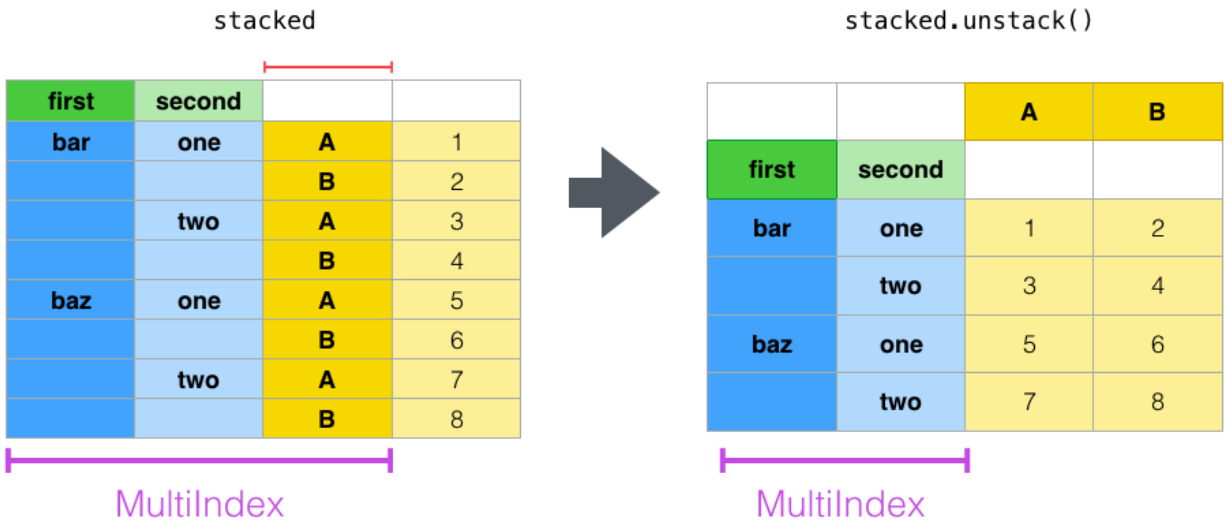
### Stack



Closely related to the `pivot()` method are the related `stack()` and `unstack()` methods available on Series and DataFrame. These methods are designed to work together with MultiIndex objects (see the section on *hierarchical indexing*). Here are essentially what these methods do:

- `stack`: “pivot” a level of the (possibly hierarchical) column labels, returning a DataFrame with an index with a new inner-most level of row labels.
- `unstack`: (inverse operation of `stack`) “pivot” a level of the (possibly hierarchical) row index to the column axis, producing a reshaped DataFrame with a new inner-most level of column labels.

# Unstack



The clearest way to explain is by example. Let's take a prior example data set from the hierarchical indexing section:

```
In [8]: tuples = list(zip(*(['bar', 'bar', 'baz', 'baz',
...:                        'foo', 'foo', 'qux', 'qux'],
...:                        ['one', 'two', 'one', 'two',
...:                        'one', 'two', 'one', 'two'])))
...:
In [9]: index = pd.MultiIndex.from_tuples(tuples, names=['first', 'second'])
In [10]: df = pd.DataFrame(np.random.randn(8, 2), index=index, columns=['A', 'B'])
In [11]: df2 = df[:4]
In [12]: df2
Out[12]:
```

|       |        | A         | B         |
|-------|--------|-----------|-----------|
| first | second |           |           |
| bar   | one    | 0.721555  | -0.706771 |
|       | two    | -1.039575 | 0.271860  |
| baz   | one    | -0.424972 | 0.567020  |
|       | two    | 0.276232  | -1.087401 |

The stack function “compresses” a level in the DataFrame’s columns to produce either:

- A Series, in the case of a simple column Index.
- A DataFrame, in the case of a MultiIndex in the columns.

If the columns have a MultiIndex, you can choose which level to stack. The stacked level becomes the new lowest level in a MultiIndex on the columns:

```
In [13]: stacked = df2.stack()
```

```
In [14]: stacked
```

Out [14]:

```
first  second
bar    one    A    0.721555
        B    -0.706771
        two    A   -1.039575
        B     0.271860
baz    one    A   -0.424972
        B     0.567020
        two    A    0.276232
        B    -1.087401
dtype: float64
```

With a “stacked” DataFrame or Series (having a MultiIndex as the index), the inverse operation of stack is `unstack`, which by default unstacks the **last level**:

```
In [15]: stacked.unstack()
```

Out [15] :

|       |        | A         | B         |
|-------|--------|-----------|-----------|
| first | second |           |           |
| bar   | one    | 0.721555  | -0.706771 |
|       | two    | -1.039575 | 0.271860  |
| baz   | one    | -0.424972 | 0.567020  |
|       | two    | 0.276232  | -1.087401 |

```
In [16]: stacked.unstack(1)
```

```

second      one      two
first
bar  A  0.721555 -1.039575
     B -0.706771  0.271860
baz  A -0.424972  0.276232
     B  0.567020 -1.087401

```

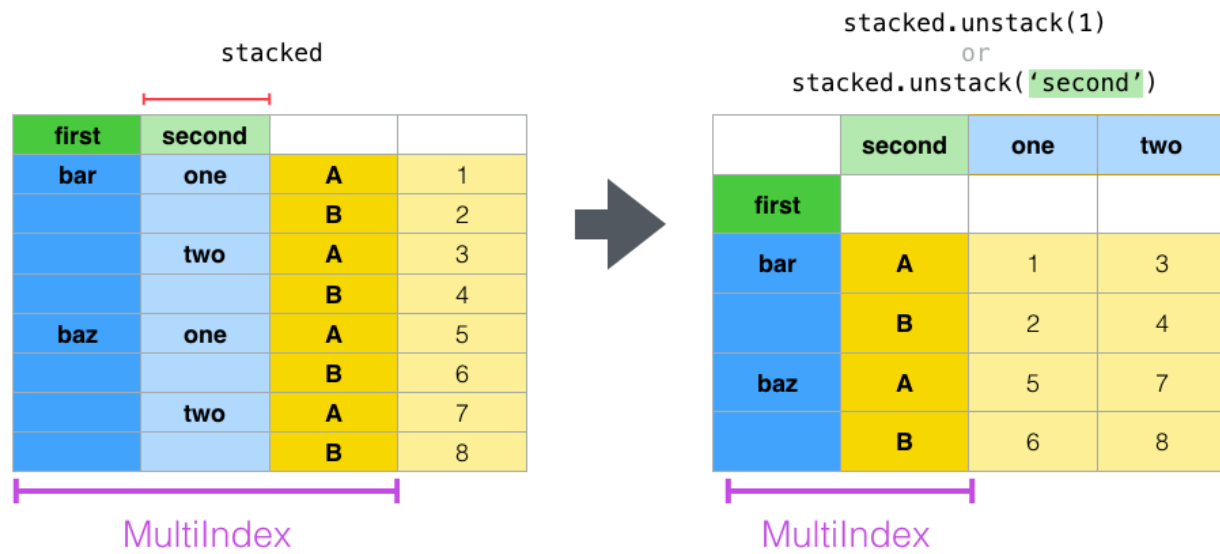
```
In [17]: stacked.unstack(0)
```

```

first      bar      baz
second
one      A      0.721555 -0.424972
          B      -0.706771  0.567020
two      A      -1.039575  0.276232
          B      0.271860 -1.087401

```

## Unstack(1)

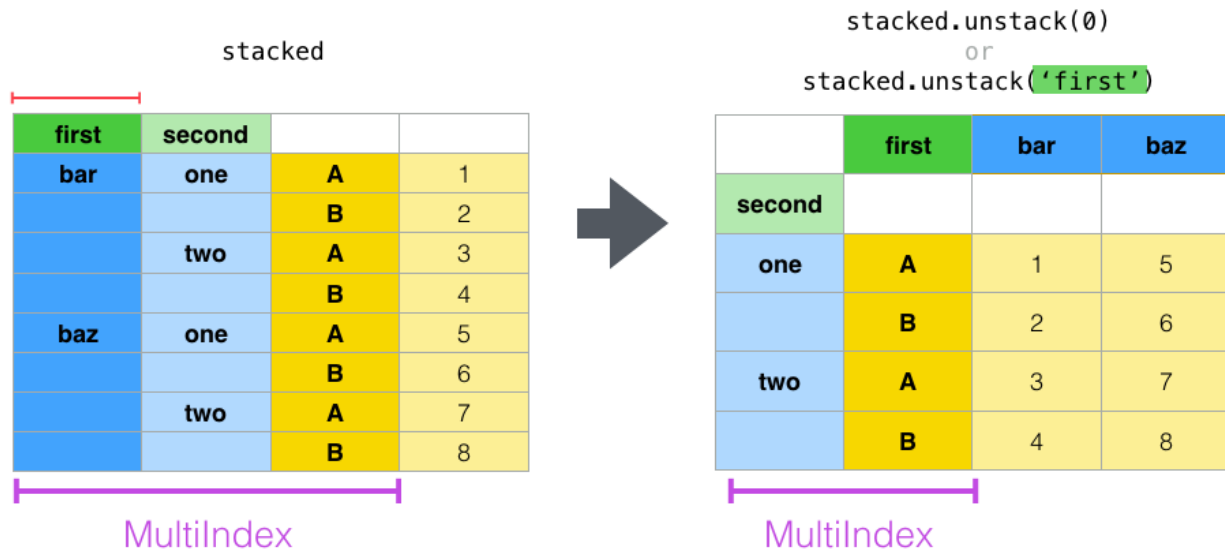


If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [18]: stacked.unstack('second')
Out[18]:
second      one      two
first
bar   A  0.721555 -1.039575
      B -0.706771  0.271860
baz   A -0.424972  0.276232
      B  0.567020 -1.087401
```



## Unstack(0)



Notice that the `stack` and `unstack` methods implicitly sort the index levels involved. Hence a call to `stack` and then `unstack`, or vice versa, will result in a **sorted** copy of the original DataFrame or Series:

```
In [19]: index = pd.MultiIndex.from_product([[2,1], ['a', 'b']])
In [20]: df = pd.DataFrame(np.random.randn(4), index=index, columns=['A'])
In [21]: df
Out[21]:
      A
2 a -0.370647
  b -1.157892
1 a -1.344312
  b  0.844885

In [22]: all(df.unstack().stack() == df.sort_index())
Out[22]:
True
```

The above code will raise a `TypeError` if the call to `sort_index` is removed.

### 18.2.1 Multiple Levels

You may also stack or unstack more than one level at a time by passing a list of levels, in which case the end result is as if each level in the list were processed individually.

```
In [23]: columns = pd.MultiIndex.from_tuples([
....:     ('A', 'cat', 'long'), ('B', 'cat', 'long'),
....:     ('A', 'dog', 'short'), ('B', 'dog', 'short')
....: ],
....:     names=['exp', 'animal', 'hair_length'])
```

(continues on next page)

(continued from previous page)

```

.....: )
.....:

In [24]: df = pd.DataFrame(np.random.randn(4, 4), columns=columns)

In [25]: df
Out[25]:
exp          A          B          A          B
animal      cat      cat      dog      dog
hair_length  long      long      short     short
0          1.075770 -0.109050  1.643563 -1.469388
1          0.357021 -0.674600 -1.776904 -0.968914
2         -1.294524  0.413738  0.276662 -0.472035
3         -0.013960 -0.362543 -0.006154 -0.923061

```

```
In [26]: df.stack(level=['animal', 'hair_length'])
```

```

////////////////////////////////////
↪
exp          A          B
animal hair_length
0 cat      long      1.075770 -0.109050
  dog      short     1.643563 -1.469388
1 cat      long      0.357021 -0.674600
  dog      short     -1.776904 -0.968914
2 cat      long     -1.294524  0.413738
  dog      short      0.276662 -0.472035
3 cat      long     -0.013960 -0.362543
  dog      short     -0.006154 -0.923061

```

The list of levels can contain either level names or level numbers (but not a mixture of the two).

```

# df.stack(level=['animal', 'hair_length'])
# from above is equivalent to:
In [27]: df.stack(level=[1, 2])

```

```

Out[27]:
exp          A          B
animal hair_length
0 cat      long      1.075770 -0.109050
  dog      short     1.643563 -1.469388
1 cat      long      0.357021 -0.674600
  dog      short     -1.776904 -0.968914
2 cat      long     -1.294524  0.413738
  dog      short      0.276662 -0.472035
3 cat      long     -0.013960 -0.362543
  dog      short     -0.006154 -0.923061

```

## 18.2.2 Missing Data

These functions are intelligent about handling missing data and do not expect each subgroup within the hierarchical index to have the same set of labels. They also can handle the index being unsorted (but you can make it sorted by calling `sort_index`, of course). Here is a more complex example:

```

In [28]: columns = pd.MultiIndex.from_tuples([('A', 'cat'), ('B', 'dog'),
.....:                                     ('B', 'cat'), ('A', 'dog')],
.....:                                     names=['exp', 'animal'])

```

(continues on next page)

(continued from previous page)

```

.....:
In [29]: index = pd.MultiIndex.from_product([('bar', 'baz', 'foo', 'qux'),
.....:                                     ('one', 'two')],
.....:                                     names=['first', 'second'])
.....:

In [30]: df = pd.DataFrame(np.random.randn(8, 4), index=index, columns=columns)

In [31]: df2 = df.iloc[[0, 1, 2, 4, 5, 7]]

In [32]: df2
Out[32]:
exp          A          B          A
animal      cat      dog      cat      dog
first second
bar   one    0.895717  0.805244 -1.206412  2.565646
      two    1.431256  1.340309 -1.170299 -0.226169
baz   one    0.410835  0.813850  0.132003 -0.827317
foo   one   -1.413681  1.607920  1.024180  0.569605
      two    0.875906 -2.211372  0.974466 -2.006747
qux   two   -1.226825  0.769804 -1.281247 -0.727707

```

As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```

In [33]: df2.stack('exp')
Out[33]:
animal      cat      dog
first second exp
bar   one    A    0.895717  2.565646
      two    B   -1.206412  0.805244
      two    A    1.431256 -0.226169
      two    B   -1.170299  1.340309
baz   one    A    0.410835 -0.827317
      two    B    0.132003  0.813850
foo   one    A   -1.413681  0.569605
      two    B    1.024180  1.607920
      two    A    0.875906 -2.006747
      two    B    0.974466 -2.211372
qux   two    A   -1.226825 -0.727707
      two    B   -1.281247  0.769804

In [34]: df2.stack('animal')
//////////
↪
exp          A          B
first second animal
bar   one    cat    0.895717 -1.206412
      two    dog    2.565646  0.805244
      two    cat    1.431256 -1.170299
      two    dog   -0.226169  1.340309
baz   one    cat    0.410835  0.132003
      two    dog   -0.827317  0.813850
foo   one    cat   -1.413681  1.024180
      two    dog    0.569605  1.607920
      two    cat    0.875906  0.974466
      two    dog   -2.006747 -2.211372

```

(continues on next page)

(continued from previous page)

|     |     |     |           |           |
|-----|-----|-----|-----------|-----------|
| qux | two | cat | -1.226825 | -1.281247 |
|     |     | dog | -0.727707 | 0.769804  |

Unstacking can result in missing values if subgroups do not have the same set of labels. By default, missing values will be replaced with the default fill value for that data type, NaN for float, NaT for datetimelike, etc. For integer types, by default data will be converted to float and missing values will be set to NaN.

```
In [35]: df3 = df.iloc[[0, 1, 4, 7], [1, 2]]
```

```
In [36]: df3
```

```
Out [36]:
```

| exp    |        | B        |           |     |
|--------|--------|----------|-----------|-----|
| animal |        | dog      |           | cat |
| first  | second |          |           |     |
| bar    | one    | 0.805244 | -1.206412 |     |
|        | two    | 1.340309 | -1.170299 |     |
| foo    | one    | 1.607920 | 1.024180  |     |
| qux    | two    | 0.769804 | -1.281247 |     |

```
In [37]: df3.unstack()
```

```

////////////////////////////////////
↪
exp      B
animal   dog      cat
second   one      two      one      two
first
bar      0.805244  1.340309 -1.206412 -1.170299
foo      1.607920      NaN  1.024180      NaN
qux      NaN      0.769804      NaN -1.281247

```

New in version 0.18.0.

Alternatively, unstack takes an optional `fill_value` argument, for specifying the value of missing data.

```
In [38]: df3.unstack(fill_value=-1e9)
```

```
Out [38]:
```

| exp    |  | B             |               |               |               |
|--------|--|---------------|---------------|---------------|---------------|
| animal |  | dog           |               | cat           |               |
| second |  | one           | two           | one           | two           |
| first  |  |               |               |               |               |
| bar    |  | 8.052440e-01  | 1.340309e+00  | -1.206412e+00 | -1.170299e+00 |
| foo    |  | 1.607920e+00  | -1.000000e+09 | 1.024180e+00  | -1.000000e+09 |
| qux    |  | -1.000000e+09 | 7.698036e-01  | -1.000000e+09 | -1.281247e+00 |

## 18.2.3 With a MultiIndex

Unstacking when the columns are a `MultiIndex` is also careful about doing the right thing:

```
In [39]: df[:3].unstack(0)
```

```
Out [39]:
```

| exp    |  | A        |          | B        |         |           |          | A        |           |
|--------|--|----------|----------|----------|---------|-----------|----------|----------|-----------|
| animal |  | cat      |          | dog      |         | cat       |          | dog      |           |
| first  |  | bar      | baz      | bar      | baz     | bar       | baz      | bar      | baz       |
| second |  |          |          |          |         |           |          |          |           |
| one    |  | 0.895717 | 0.410835 | 0.805244 | 0.81385 | -1.206412 | 0.132003 | 2.565646 | -0.827317 |

(continues on next page)

(continued from previous page)

```

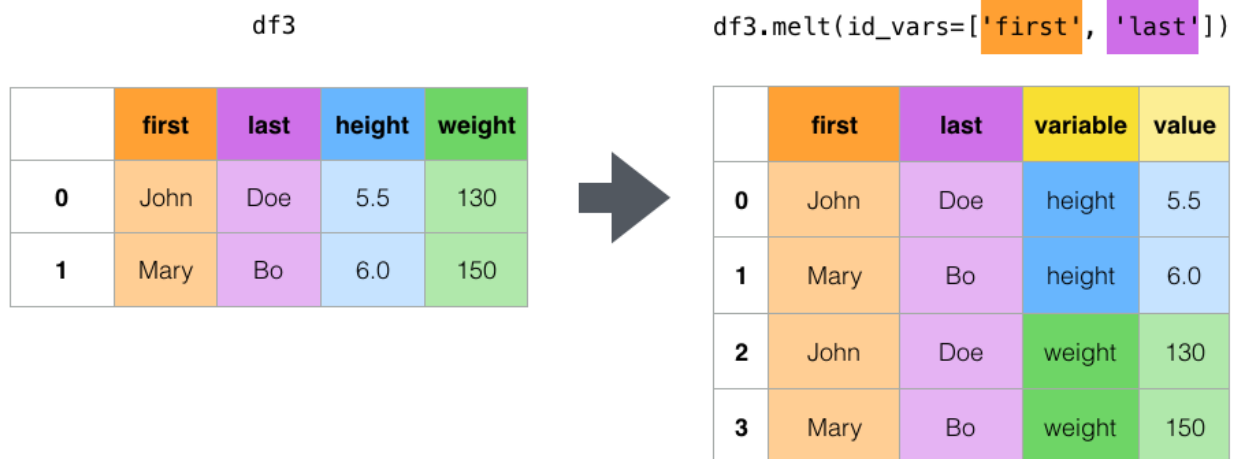
two      1.431256      NaN  1.340309      NaN -1.170299      NaN -0.226169      NaN

In [40]: df2.unstack(1)
//////////
↪
exp      A      B      A
animal   cat   dog   dog
second  one   two   one   two   one   two   one   two
first
bar      0.895717  1.431256  0.805244  1.340309 -1.206412 -1.170299  2.565646 -0.226169
baz      0.410835      NaN  0.813850      NaN  0.132003      NaN -0.827317      NaN
foo     -1.413681  0.875906  1.607920 -2.211372  1.024180  0.974466  0.569605 -2.006747
qux      NaN -1.226825      NaN  0.769804      NaN -1.281247      NaN -0.727707

```

## 18.3 Reshaping by Melt

### Melt



The top-level `melt()` function and the corresponding `DataFrame.melt()` are useful to massage a `DataFrame` into a format where one or more columns are *identifier variables*, while all other columns, considered *measured variables*, are “unpivoted” to the row axis, leaving just two non-identifier columns, “variable” and “value”. The names of those columns can be customized by supplying the `var_name` and `value_name` parameters.

For instance,

```

In [41]: cheese = pd.DataFrame({'first' : ['John', 'Mary'],
.....:                        'last'  : ['Doe', 'Bo'],
.....:                        'height' : [5.5, 6.0],
.....:                        'weight' : [130, 150]})
.....:

In [42]: cheese
Out [42]:

```

(continues on next page)

(continued from previous page)

```

    first last  height  weight
0   John   Doe     5.5    130
1   Mary    Bo     6.0    150

In [43]: cheese.melt(id_vars=['first', 'last'])
Out[43]:
↪
   first last  variable  value
0   John   Doe    height     5.5
1   Mary    Bo    height     6.0
2   John   Doe   weight   130.0
3   Mary    Bo   weight   150.0

In [44]: cheese.melt(id_vars=['first', 'last'], var_name='quantity')
Out[44]:
↪
   first last quantity  value
0   John   Doe    height     5.5
1   Mary    Bo    height     6.0
2   John   Doe   weight   130.0
3   Mary    Bo   weight   150.0

```

Another way to transform is to use the `wide_to_long()` panel data convenience function. It is less flexible than `melt()`, but more user-friendly.

```

In [45]: dft = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
.....:                      "A1980" : {0 : "d", 1 : "e", 2 : "f"},
.....:                      "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
.....:                      "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
.....:                      "X"      : dict(zip(range(3), np.random.randn(3)))
.....:                      })

In [46]: dft["id"] = dft.index

In [47]: dft
Out[47]:
   A1970 A1980  B1970  B1980         X  id
0      a      d    2.5    3.2 -0.121306  0
1      b      e    1.2    1.3 -0.097883  1
2      c      f    0.7    0.1  0.695775  2

In [48]: pd.wide_to_long(dft, ["A", "B"], i="id", j="year")
Out[48]:
↪
           X  A    B
id year
0  1970 -0.121306  a  2.5
1  1970 -0.097883  b  1.2
2  1970  0.695775  c  0.7
0  1980 -0.121306  d  3.2
1  1980 -0.097883  e  1.3
2  1980  0.695775  f  0.1

```

## 18.4 Combining with stats and GroupBy

It should be no shock that combining `pivot / stack / unstack` with `GroupBy` and the basic `Series` and `DataFrame` statistical functions can produce some very expressive and fast data manipulations.

```
In [49]: df
Out[49]:
```

|        |        | A         |           | B         |           |
|--------|--------|-----------|-----------|-----------|-----------|
| exp    |        | cat       | dog       | cat       | dog       |
| animal |        |           |           |           |           |
| first  | second |           |           |           |           |
| bar    | one    | 0.895717  | 0.805244  | -1.206412 | 2.565646  |
|        | two    | 1.431256  | 1.340309  | -1.170299 | -0.226169 |
| baz    | one    | 0.410835  | 0.813850  | 0.132003  | -0.827317 |
|        | two    | -0.076467 | -1.187678 | 1.130127  | -1.436737 |
| foo    | one    | -1.413681 | 1.607920  | 1.024180  | 0.569605  |
|        | two    | 0.875906  | -2.211372 | 0.974466  | -2.006747 |
| qux    | one    | -0.410001 | -0.078638 | 0.545952  | -1.219217 |
|        | two    | -1.226825 | 0.769804  | -1.281247 | -0.727707 |

```
In [50]: df.stack().mean(1).unstack()
```

```

////////////////////////////////////
↪
animal      cat      dog
first second
bar  one  -0.155347  1.685445
     two   0.130479  0.557070
baz   one   0.271419 -0.006733
     two   0.526830 -1.312207
foo   one  -0.194750  1.088763
     two   0.925186 -2.109060
qux   one   0.067976 -0.648927
     two  -1.254036  0.021048

```

```
# same result, another way
```

```
In [51]: df.groupby(level=1, axis=1).mean()
```

```

////////////////////////////////////
↪
animal      cat      dog
first second
bar  one  -0.155347  1.685445
     two   0.130479  0.557070
baz   one   0.271419 -0.006733
     two   0.526830 -1.312207
foo   one  -0.194750  1.088763
     two   0.925186 -2.109060
qux   one   0.067976 -0.648927
     two  -1.254036  0.021048

```

```
In [52]: df.stack().groupby(level=1).mean()
```

```

////////////////////////////////////
↪
exp      A      B
second
one    0.071448  0.455513
two   -0.424186 -0.204486

```

```
In [53]: df.mean().unstack(0)
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
exp          A          B
animal
cat      0.060843  0.018596
dog      -0.413580  0.232430

```

## 18.5 Pivot tables

While `pivot()` provides general purpose pivoting with various data types (strings, numerics, etc.), pandas also provides `pivot_table()` for pivoting with aggregation of numeric data.

The function `pivot_table()` can be used to create spreadsheet-style pivot tables. See the [cookbook](#) for some advanced strategies.

It takes a number of arguments:

- `data`: a `DataFrame` object.
- `values`: a column or a list of columns to aggregate.
- `index`: a column, `Grouper`, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
- `columns`: a column, `Grouper`, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
- `aggfunc`: function to use for aggregation, defaulting to `numpy.mean`.

Consider a data set like this:

```

In [54]: import datetime

In [55]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 6,
.....:                    'B': ['A', 'B', 'C'] * 8,
.....:                    'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 4,
.....:                    'D': np.random.randn(24),
.....:                    'E': np.random.randn(24),
.....:                    'F': [datetime.datetime(2013, i, 1) for i in range(1,
↪13)] +
.....:                    [datetime.datetime(2013, i, 15) for i in range(1,
↪13)]})

In [56]: df
Out[56]:
   A  B  C         D         E         F
0  one A  foo  0.341734 -0.317441 2013-01-01
1  one B  foo  0.959726 -1.236269 2013-02-01
2  two C  foo -1.110336  0.896171 2013-03-01
3 three A  bar -0.619976 -0.487602 2013-04-01
4  one B  bar  0.149748 -0.082240 2013-05-01
5  one C  bar -0.732339 -2.182937 2013-06-01
6  two A  foo  0.687738  0.380396 2013-07-01
..  ..  ..  ...  ...  ...
17 one C  bar -0.345352  0.206053 2013-06-15
18 two A  foo  1.314232 -0.251905 2013-07-15

```

(continues on next page)



(continued from previous page)

```

19  three  B  foo  0.690579 -2.213588 2013-08-15
20   one   C  foo  0.995761  1.063327 2013-09-15
21   one   A  bar  2.396780  1.266143 2013-10-15
22   two   B  bar  0.014871  0.299368 2013-11-15
23  three   C  bar  3.357427 -0.863838 2013-12-15

```

```
[24 rows x 6 columns]
```

We can produce pivot tables from this data very easily:

```
In [57]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
```

```
Out[57]:
```

```

C          bar          foo
A  B
one A  1.120915 -0.514058
    B -0.338421  0.002759
    C -0.538846  0.699535
three A -1.181568      NaN
      B      NaN  0.433512
      C  0.588783      NaN
two   A      NaN  1.000985
      B  0.158248      NaN
      C      NaN  0.176180

```

```
In [58]: pd.pivot_table(df, values='D', index=['B'], columns=['A', 'C'], aggfunc=np.
→sum)
```

```

////////////////////////////////////
→
A      one          three          two
C      bar          foo          bar          foo          bar          foo
B
A  2.241830 -1.028115 -2.363137      NaN      NaN  2.001971
B -0.676843  0.005518      NaN  0.867024  0.316495      NaN
C -1.077692  1.399070  1.177566      NaN      NaN  0.352360

```

```
In [59]: pd.pivot_table(df, values=['D', 'E'], index=['B'], columns=['A', 'C'],
→aggfunc=np.sum)
```

```

////////////////////////////////////
→
      D          ...          E
→
A      one          three          two          ...          one          three
→
C      bar          foo          bar          foo          bar          ...          foo          bar
→  foo          bar          foo
B
→
A  2.241830 -1.028115 -2.363137      NaN      NaN          ... -0.043211  1.922577
→  NaN          NaN  0.128491
B -0.676843  0.005518      NaN  0.867024  0.316495          ... -1.103384      NaN -2.
→128743 -0.194294      NaN
C -1.077692  1.399070  1.177566      NaN      NaN          ...  1.495717 -0.263660
→  NaN          NaN  0.872482

```

```
[3 rows x 12 columns]
```

The result object is a DataFrame having potentially hierarchical indexes on the rows and columns. If the values

column name is not given, the pivot table will include all of the data that can be aggregated in an additional level of hierarchy in the columns:

```
In [60]: pd.pivot_table(df, index=['A', 'B'], columns=['C'])
```

```
Out [60]:
```

|       |   | D         |           | E         |           |
|-------|---|-----------|-----------|-----------|-----------|
| C     |   | bar       | foo       | bar       | foo       |
| one   | A | 1.120915  | -0.514058 | 1.393057  | -0.021605 |
|       | B | -0.338421 | 0.002759  | 0.684140  | -0.551692 |
|       | C | -0.538846 | 0.699535  | -0.988442 | 0.747859  |
| three | A | -1.181568 | NaN       | 0.961289  | NaN       |
|       | B | NaN       | 0.433512  | NaN       | -1.064372 |
|       | C | 0.588783  | NaN       | -0.131830 | NaN       |
| two   | A | NaN       | 1.000985  | NaN       | 0.064245  |
|       | B | 0.158248  | NaN       | -0.097147 | NaN       |
|       | C | NaN       | 0.176180  | NaN       | 0.436241  |

Also, you can use `Grouper` for index and columns keywords. For detail of `Grouper`, see [Grouping with a Grouper specification](#).

```
In [61]: pd.pivot_table(df, values='D', index=pd.Grouper(freq='M', key='F'), columns=
→ 'C')
```

```
Out [61]:
```

| C          |  | bar       | foo       |
|------------|--|-----------|-----------|
| F          |  |           |           |
| 2013-01-31 |  | NaN       | -0.514058 |
| 2013-02-28 |  | NaN       | 0.002759  |
| 2013-03-31 |  | NaN       | 0.176180  |
| 2013-04-30 |  | -1.181568 | NaN       |
| 2013-05-31 |  | -0.338421 | NaN       |
| 2013-06-30 |  | -0.538846 | NaN       |
| 2013-07-31 |  | NaN       | 1.000985  |
| 2013-08-31 |  | NaN       | 0.433512  |
| 2013-09-30 |  | NaN       | 0.699535  |
| 2013-10-31 |  | 1.120915  | NaN       |
| 2013-11-30 |  | 0.158248  | NaN       |
| 2013-12-31 |  | 0.588783  | NaN       |

You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```
In [62]: table = pd.pivot_table(df, index=['A', 'B'], columns=['C'])
```

```
In [63]: print(table.to_string(na_rep=''))
```

|       |   | D         |           | E         |           |
|-------|---|-----------|-----------|-----------|-----------|
| C     |   | bar       | foo       | bar       | foo       |
| one   | A | 1.120915  | -0.514058 | 1.393057  | -0.021605 |
|       | B | -0.338421 | 0.002759  | 0.684140  | -0.551692 |
|       | C | -0.538846 | 0.699535  | -0.988442 | 0.747859  |
| three | A | -1.181568 |           | 0.961289  |           |
|       | B |           | 0.433512  |           | -1.064372 |
|       | C | 0.588783  |           | -0.131830 |           |
| two   | A |           | 1.000985  |           | 0.064245  |
|       | B | 0.158248  |           | -0.097147 |           |
|       | C |           | 0.176180  |           | 0.436241  |

Note that `pivot_table` is also available as an instance method on `DataFrame`, i.e. `DataFrame.pivot_table()`.

*DataFrame.*

### 18.5.1 Adding margins

If you pass `margins=True` to `pivot_table`, special All columns and rows will be added with partial group aggregates across the categories on the rows and columns:

```
In [64]: df.pivot_table(index=['A', 'B'], columns='C', margins=True, aggfunc=np.std)
Out [64]:
```

|       |   | D        |          |          | E        |          |          |
|-------|---|----------|----------|----------|----------|----------|----------|
| C     |   | bar      | foo      | All      | bar      | foo      | All      |
| one   | A | 1.804346 | 1.210272 | 1.569879 | 0.179483 | 0.418374 | 0.858005 |
|       | B | 0.690376 | 1.353355 | 0.898998 | 1.083825 | 0.968138 | 1.101401 |
|       | C | 0.273641 | 0.418926 | 0.771139 | 1.689271 | 0.446140 | 1.422136 |
| three | A | 0.794212 | NaN      | 0.794212 | 2.049040 | NaN      | 2.049040 |
|       | B | NaN      | 0.363548 | 0.363548 | NaN      | 1.625237 | 1.625237 |
|       | C | 3.915454 | NaN      | 3.915454 | 1.035215 | NaN      | 1.035215 |
| two   | A | NaN      | 0.442998 | 0.442998 | NaN      | 0.447104 | 0.447104 |
|       | B | 0.202765 | NaN      | 0.202765 | 0.560757 | NaN      | 0.560757 |
|       | C | NaN      | 1.819408 | 1.819408 | NaN      | 0.650439 | 0.650439 |
| All   |   | 1.556686 | 0.952552 | 1.246608 | 1.250924 | 0.899904 | 1.059389 |

## 18.6 Cross tabulations

Use `crosstab()` to compute a cross-tabulation of two (or more) factors. By default `crosstab` computes a frequency table of the factors unless an array of values and an aggregation function are passed.

It takes a number of arguments

- `index`: array-like, values to group by in the rows.
- `columns`: array-like, values to group by in the columns.
- `values`: array-like, optional, array of values to aggregate according to the factors.
- `aggfunc`: function, optional, If no values array is passed, computes a frequency table.
- `rownames`: sequence, default `None`, must match number of row arrays passed.
- `colnames`: sequence, default `None`, if passed, must match number of column arrays passed.
- `margins`: boolean, default `False`, Add row/column margins (subtotals)
- `normalize`: boolean, {'all', 'index', 'columns'}, or {0,1}, default `False`. Normalize by dividing all values by the sum of values.

Any `Series` passed will have their name attributes used unless row or column names for the cross-tabulation are specified

For example:

```
In [65]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'

In [66]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)

In [67]: b = np.array([one, one, two, one, two, one], dtype=object)

In [68]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)

In [69]: pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
```

(continues on next page)

(continued from previous page)

```
Out [69]:
b      one      two
c  dull shiny dull shiny
a
bar      1      0      0      1
foo      2      1      1      0
```

If `crosstab` receives only two Series, it will provide a frequency table.

```
In [70]: df = pd.DataFrame({'A': [1, 2, 2, 2, 2], 'B': [3, 3, 4, 4, 4],
.....:                    'C': [1, 1, np.nan, 1, 1]})
.....:
```

```
In [71]: df
```

```
Out [71]:
   A  B    C
0  1  3  1.0
1  2  3  1.0
2  2  4  NaN
3  2  4  1.0
4  2  4  1.0
```

```
In [72]: pd.crosstab(df.A, df.B)
```

```
Out [72]:
B  3  4
A
1  1  0
2  1  3
```

Any input passed containing Categorical data will have **all** of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

```
In [73]: foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])
```

```
In [74]: bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])
```

```
In [75]: pd.crosstab(foo, bar)
```

```
Out [75]:
col_0  d  e
row_0
a      1  0
b      0  1
```

## 18.6.1 Normalization

New in version 0.18.1.

Frequency tables can also be normalized to show percentages rather than counts using the `normalize` argument:

```
In [76]: pd.crosstab(df.A, df.B, normalize=True)
```

```
Out [76]:
B      3      4
A
1  0.2  0.0
2  0.2  0.6
```

`normalize` can also normalize values within each row or within each column:

```
In [77]: pd.crosstab(df.A, df.B, normalize='columns')
Out[77]:
B      3      4
A
1    0.5    0.0
2    0.5    1.0
```

`crosstab` can also be passed a third Series and an aggregation function (`aggfunc`) that will be applied to the values of the third Series within each group defined by the first two Series:

```
In [78]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum)
Out[78]:
B      3      4
A
1    1.0    NaN
2    1.0    2.0
```

## 18.6.2 Adding Margins

Finally, one can also add margins or normalize this output.

```
In [79]: pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum, normalize=True,
.....:               margins=True)
Out[79]:
B      3      4    All
A
1    0.25  0.0  0.25
2    0.25  0.5  0.75
All  0.50  0.5  1.00
```

## 18.7 Tiling

The `cut()` function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables:

```
In [80]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])

In [81]: pd.cut(ages, bins=3)
Out[81]:
[(9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.95, 26.667], (9.
→95, 26.667], (26.667, 43.333], (43.333, 60.0], (43.333, 60.0]]
Categories (3, interval[float64]): [(9.95, 26.667] < (26.667, 43.333] < (43.333, 60.
→0]]
```

If the `bins` keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

```
In [82]: c = pd.cut(ages, bins=[0, 18, 35, 70])

In [83]: c
Out[83]:
```

(continues on next page)

(continued from previous page)

```
[ (0, 18], (0, 18], (0, 18], (0, 18], (18, 35], (18, 35], (18, 35], (35, 70], (35, 70] ]
Categories (3, interval[int64]): [(0, 18] < (18, 35] < (35, 70]]
```

## New in version 0.20.0.

If the `bins` keyword is an `IntervalIndex`, then these will be used to bin the passed data.:

```
pd.cut([25, 20, 50], bins=c.categories)
```

## 18.8 Computing indicator / dummy variables

To convert a categorical variable into a “dummy” or “indicator” DataFrame, for example a column in a DataFrame (a Series) which has k distinct values, can derive a DataFrame containing k columns of 1s and 0s using `get_dummies()`:

```
In [84]: df = pd.DataFrame({'key': list('bbacab'), 'data1': range(6)})
```

```
In [85]: pd.get_dummies(df['key'])
```

Out [85] :

|   | a | b | c |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 0 |
| 3 | 0 | 0 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 |

Sometimes it's useful to prefix the column names, for example when merging the result with the original `DataFrame`:

```
In [86]: dummies = pd.get_dummies(df['key'], prefix='key')
```

```
In [87]: dummies
```

Out [87] :

|   | key_a | key_b | key_c |
|---|-------|-------|-------|
| 0 | 0     | 1     | 0     |
| 1 | 0     | 1     | 0     |
| 2 | 1     | 0     | 0     |
| 3 | 0     | 0     | 1     |
| 4 | 1     | 0     | 0     |
| 5 | 0     | 1     | 0     |

```
In [88]: df[['data1']].join(dummies)
```

|   | data1 | key_a | key_b | key_c |
|---|-------|-------|-------|-------|
| 0 | 0     | 0     | 1     | 0     |
| 1 | 1     | 0     | 1     | 0     |
| 2 | 2     | 1     | 0     | 0     |
| 3 | 3     | 0     | 0     | 1     |
| 4 | 4     | 1     | 0     | 0     |
| 5 | 5     | 0     | 1     | 0     |

This function is often used along with discretization functions like `cut`:

```
In [89]: values = np.random.randn(10)

In [90]: values
Out[90]:
array([ 0.4082, -1.0481, -0.0257, -0.9884,  0.0941,  1.2627,  1.29   ,
        0.0824, -0.0558,  0.5366])

In [91]: bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

In [92]: pd.get_dummies(pd.cut(values, bins))
Out[92]:
```

|   | (0.0, 0.2] | (0.2, 0.4] | (0.4, 0.6] | (0.6, 0.8] | (0.8, 1.0] |
|---|------------|------------|------------|------------|------------|
| 0 | 0          | 0          | 1          | 0          | 0          |
| 1 | 0          | 0          | 0          | 0          | 0          |
| 2 | 0          | 0          | 0          | 0          | 0          |
| 3 | 0          | 0          | 0          | 0          | 0          |
| 4 | 1          | 0          | 0          | 0          | 0          |
| 5 | 0          | 0          | 0          | 0          | 0          |
| 6 | 0          | 0          | 0          | 0          | 0          |
| 7 | 1          | 0          | 0          | 0          | 0          |
| 8 | 0          | 0          | 0          | 0          | 0          |
| 9 | 0          | 0          | 1          | 0          | 0          |

See also `Series.str.get_dummies`.

`get_dummies()` also accepts a `DataFrame`. By default all categorical variables (categorical in the statistical sense, those with `object` or `categorical` dtype) are encoded as dummy variables.

```
In [93]: df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
.....:                    'C': [1, 2, 3]})
.....:

In [94]: pd.get_dummies(df)
Out[94]:
```

|   | C | A_a | A_b | B_b | B_c |
|---|---|-----|-----|-----|-----|
| 0 | 1 | 1   | 0   | 0   | 1   |
| 1 | 2 | 0   | 1   | 0   | 1   |
| 2 | 3 | 1   | 0   | 1   | 0   |

All non-object columns are included untouched in the output. You can control the columns that are encoded with the `columns` keyword.

```
In [95]: pd.get_dummies(df, columns=['A'])
Out[95]:
```

|   | B | C | A_a | A_b |
|---|---|---|-----|-----|
| 0 | c | 1 | 1   | 0   |
| 1 | c | 2 | 0   | 1   |
| 2 | b | 3 | 1   | 0   |

Notice that the `B` column is still included in the output, it just hasn't been encoded. You can drop `B` before calling `get_dummies` if you don't want to include it in the output.

As with the `Series` version, you can pass values for the `prefix` and `prefix_sep`. By default the column name is used as the prefix, and `'_'` as the prefix separator. You can specify `prefix` and `prefix_sep` in 3 ways:

- string: Use the same value for `prefix` or `prefix_sep` for each column to be encoded.
- list: Must be the same length as the number of columns being encoded.
- dict: Mapping column name to prefix.

```
In [96]: simple = pd.get_dummies(df, prefix='new_prefix')

In [97]: simple
Out[97]:
```

|   | C | new_prefix_a | new_prefix_b | new_prefix_b | new_prefix_c |
|---|---|--------------|--------------|--------------|--------------|
| 0 | 1 | 1            | 0            | 0            | 1            |
| 1 | 2 | 0            | 1            | 0            | 1            |
| 2 | 3 | 1            | 0            | 1            | 0            |

```
In [98]: from_list = pd.get_dummies(df, prefix=['from_A', 'from_B'])

In [99]: from_list
Out[99]:
```

|   | C | from_A_a | from_A_b | from_B_b | from_B_c |
|---|---|----------|----------|----------|----------|
| 0 | 1 | 1        | 0        | 0        | 1        |
| 1 | 2 | 0        | 1        | 0        | 1        |
| 2 | 3 | 1        | 0        | 1        | 0        |

```
In [100]: from_dict = pd.get_dummies(df, prefix={'B': 'from_B', 'A': 'from_A'})

In [101]: from_dict
Out[101]:
```

|   | C | from_A_a | from_A_b | from_B_b | from_B_c |
|---|---|----------|----------|----------|----------|
| 0 | 1 | 1        | 0        | 0        | 1        |
| 1 | 2 | 0        | 1        | 0        | 1        |
| 2 | 3 | 1        | 0        | 1        | 0        |

New in version 0.18.0.

Sometimes it will be useful to only keep k-1 levels of a categorical variable to avoid collinearity when feeding the result to statistical models. You can switch to this mode by turn on `drop_first`.

```
In [102]: s = pd.Series(list('abcaa'))

In [103]: pd.get_dummies(s)
Out[103]:
```

|   | a | b | c |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 1 | 0 | 0 |
| 4 | 1 | 0 | 0 |

```
In [104]: pd.get_dummies(s, drop_first=True)
Out[104]:
```

|   | b | c |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 0 |
| 2 | 0 | 1 |
| 3 | 0 | 0 |
| 4 | 0 | 0 |

When a column contains only one level, it will be omitted in the result.

```
In [105]: df = pd.DataFrame({'A':list('aaaaa'), 'B':list('ababc')})

In [106]: pd.get_dummies(df)
```

(continues on next page)



(continued from previous page)

**Out [106]:**

|   | A_a | B_a | B_b | B_c |
|---|-----|-----|-----|-----|
| 0 | 1   | 1   | 0   | 0   |
| 1 | 1   | 0   | 1   | 0   |
| 2 | 1   | 1   | 0   | 0   |
| 3 | 1   | 0   | 1   | 0   |
| 4 | 1   | 0   | 0   | 1   |

**In [107]:** `pd.get_dummies(df, drop_first=True)`

```

////////////////////////////////////
→
   B_b  B_c
0    0    0
1    1    0
2    0    0
3    1    0
4    0    1

```

By default new columns will have `np.uint8` dtype. To choose another dtype, use the “dtype” argument:

**In [108]:** `df = pd.DataFrame({'A': list('abc'), 'B': [1.1, 2.2, 3.3]})`**In [109]:** `pd.get_dummies(df, dtype=bool).dtypes`**Out [109]:**

```

B          float64
A_a         bool
A_b         bool
A_c         bool
dtype: object

```

New in version 0.23.0.

## 18.9 Factorizing values

To encode 1-d values as an enumerated type use `factorize()`:

**In [110]:** `x = pd.Series(['A', 'A', np.nan, 'B', 3.14, np.inf])`**In [111]:** `x`**Out [111]:**

```

0      A
1      A
2    NaN
3      B
4    3.14
5     inf
dtype: object

```

**In [112]:** `labels, uniques = pd.factorize(x)`**In [113]:** `labels`**Out [113]:** `array([ 0, 0, -1, 1, 2, 3])`**In [114]:** `uniques`

(continues on next page)

(continued from previous page)

```
Out[114]: Index(['A', 'B', 3.14, inf],  
dtype='object')
```

Note that `factorize` is similar to `numpy.unique`, but differs in its handling of NaN:

**Note:** The following `numpy.unique` will fail under Python 3 with a `TypeError` because of an ordering bug. See also [here](#).

```
In [2]: pd.factorize(x, sort=True)
Out[2]:
(array([ 2,  2, -1,  3,  0,  1]),
 Index([3.14, inf, u'A', u'B'], dtype='object'))

In [3]: np.unique(x, return_inverse=True)[::-1]
Out[3]: (array([3, 3, 0, 4, 1, 2]), array([nan, 3.14, inf, 'A', 'B'], dtype=object))
```

**Note:** If you just want to handle one column as a categorical variable (like R's factor), you can use `df["cat_col"] = pd.Categorical(df["col"])` or `df["cat_col"] = df["col"].astype("category")`. For full docs on *Categorical*, see the *Categorical introduction* and the *API documentation*.

## TIME SERIES / DATE FUNCTIONALITY

pandas has proven very successful as a tool for working with time series data, especially in the financial data analysis space. Using the NumPy `datetime64` and `timedelta64` dtypes, we have consolidated a large number of features from other Python libraries like `scikits.timeseries` as well as created a tremendous amount of new functionality for manipulating time series data.

In working with time series data, we will frequently seek to:

- generate sequences of fixed-frequency dates and time spans
- conform or convert time series to a particular frequency
- compute “relative” dates based on various non-standard time increments (e.g. 5 business days before the last business day of the year), or “roll” dates forward or backward

pandas provides a relatively compact and self-contained set of tools for performing the above tasks.

Create a range of dates:

```
# 72 hours starting with midnight Jan 1st, 2011
In [1]: rng = pd.date_range('1/1/2011', periods=72, freq='H')

In [2]: rng[:5]
Out[2]:
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 01:00:00',
               '2011-01-01 02:00:00', '2011-01-01 03:00:00',
               '2011-01-01 04:00:00'],
              dtype='datetime64[ns]', freq='H')
```

Index pandas objects with dates:

```
In [3]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [4]: ts.head()
Out[4]:
2011-01-01 00:00:00    0.469112
2011-01-01 01:00:00   -0.282863
2011-01-01 02:00:00   -1.509059
2011-01-01 03:00:00   -1.135632
2011-01-01 04:00:00    1.212112
Freq: H, dtype: float64
```

Change frequency and fill gaps:

```
# to 45 minute frequency and forward fill
In [5]: converted = ts.asfreq('45Min', method='pad')
```

(continues on next page)

(continued from previous page)

```
In [6]: converted.head()
Out [6]:
2011-01-01 00:00:00    0.469112
2011-01-01 00:45:00    0.469112
2011-01-01 01:30:00   -0.282863
2011-01-01 02:15:00   -1.509059
2011-01-01 03:00:00   -1.135632
Freq: 45T, dtype: float64
```

Resample the series to a daily frequency:

```
# Daily means
In [7]: ts.resample('D').mean()
Out [7]:
2011-01-01    -0.319569
2011-01-02    -0.337703
2011-01-03     0.117258
Freq: D, dtype: float64
```

## 19.1 Overview

The following table shows the type of time-related classes pandas can handle and how to create them.

| Class         | Remarks                        | How to create                                       |
|---------------|--------------------------------|---|
| Timestamp     | Represents a single times-tamp | to_datetime, Timestamp                              |
| DatetimeIndex | Index of Timestamp             | to_datetime, date_range, bdate_range, DatetimeIndex |
| Period        | Represents a single time span  | Period  |
| PeriodIndex   | Index of Period                | period_range, PeriodIndex                           |

## 19.2 Timestamps vs. Time Spans

Timestamped data is the most basic type of time series data that associates values with points in time. For pandas objects it means using the points in time.

```
In [8]: pd.Timestamp(datetime(2012, 5, 1))
Out [8]: Timestamp('2012-05-01 00:00:00')

In [9]: pd.Timestamp('2012-05-01')
Out [9]: Timestamp('2012-05-01 00:00:00')

In [10]: pd.Timestamp(2012, 5, 1)
Out [10]: Timestamp('2012-05-01 00:00:00')
```

However, in many cases it is more natural to associate things like change variables with a time span instead. The span represented by `Period` can be specified explicitly, or inferred from datetime string format.

For example:

```
In [11]: pd.Period('2011-01')
Out[11]: Period('2011-01', 'M')

In [12]: pd.Period('2012-05', freq='D')
Out[12]: Period('2012-05-01', 'D')
```

*Timestamp* and *Period* can serve as an index. Lists of *Timestamp* and *Period* are automatically coerced to *DatetimeIndex* and *PeriodIndex* respectively.

```
In [13]: dates = [pd.Timestamp('2012-05-01'), pd.Timestamp('2012-05-02'), pd.
↳Timestamp('2012-05-03')]

In [14]: ts = pd.Series(np.random.randn(3), dates)

In [15]: type(ts.index)
Out[15]: pandas.core.indexes.datetimes.DatetimeIndex

In [16]: ts.index
Out[16]: DatetimeIndex(['2012-05-
↳01', '2012-05-02', '2012-05-03'], dtype='datetime64[ns]', freq=None)

In [17]: ts
Out[17]:
2012-05-01    -0.410001
2012-05-02    -0.078638
2012-05-03     0.545952
dtype: float64

In [18]: periods = [pd.Period('2012-01'), pd.Period('2012-02'), pd.Period('2012-03')]

In [19]: ts = pd.Series(np.random.randn(3), periods)

In [20]: type(ts.index)
Out[20]: pandas.core.indexes.period.PeriodIndex

In [21]: ts.index
Out[21]: PeriodIndex(['2012-01',
↳'2012-02', '2012-03'], dtype='period[M]', freq='M')

In [22]: ts
Out[22]:
2012-01    -1.219217
2012-02    -1.226825
2012-03     0.769804
Freq: M, dtype: float64
```

pandas allows you to capture both representations and convert between them. Under the hood, pandas represents timestamps using instances of *Timestamp* and sequences of timestamps using instances of *DatetimeIndex*. For regular time spans, pandas uses *Period* objects for scalar values and *PeriodIndex* for sequences of spans. Better support for irregular intervals with arbitrary start and end points are forth-coming in future releases.

## 19.3 Converting to Timestamps

To convert a *Series* or list-like object of date-like objects e.g. strings, epochs, or a mixture, you can use the `to_datetime` function. When passed a *Series*, this returns a *Series* (with the same index), while a list-like is converted to a *DatetimeIndex*:

```
In [23]: pd.to_datetime(pd.Series(['Jul 31, 2009', '2010-01-10', None]))
Out[23]:
0    2009-07-31
1    2010-01-10
2         NaT
dtype: datetime64[ns]

In [24]: pd.to_datetime(['2005/11/23', '2010.12.31'])
Out[24]:
DatetimeIndex(['2005-11-23', '2010-12-31'], dtype='datetime64[ns]', freq=None)
```

If you use dates which start with the day first (i.e. European style), you can pass the `dayfirst` flag:

```
In [25]: pd.to_datetime(['04-01-2012 10:00'], dayfirst=True)
Out[25]: DatetimeIndex(['2012-01-04 10:00:00'], dtype='datetime64[ns]', freq=None)

In [26]: pd.to_datetime(['14-01-2012', '01-14-2012'], dayfirst=True)
Out[26]:
DatetimeIndex(['2012-01-14', '2012-01-14'], dtype='datetime64[ns]', freq=None)
```

**Warning:** You see in the above example that `dayfirst` isn't strict, so if a date can't be parsed with the day being first it will be parsed as if `dayfirst` were `False`.

If you pass a single string to `to_datetime`, it returns a single *Timestamp*. *Timestamp* can also accept string input, but it doesn't accept string parsing options like `dayfirst` or `format`, so use `to_datetime` if these are required.

```
In [27]: pd.to_datetime('2010/11/12')
Out[27]: Timestamp('2010-11-12 00:00:00')

In [28]: pd.Timestamp('2010/11/12')
Out[28]: Timestamp('2010-11-12 00:00:00')
```

### 19.3.1 Providing a Format Argument

In addition to the required datetime string, a `format` argument can be passed to ensure specific parsing. This could also potentially speed up the conversion considerably.

```
In [29]: pd.to_datetime('2010/11/12', format='%Y/%m/%d')
Out[29]: Timestamp('2010-11-12 00:00:00')

In [30]: pd.to_datetime('12-11-2010 00:00', format='%d-%m-%Y %H:%M')
Out[30]: Timestamp('2010-11-12 00:00:00')
```

For more information on the choices available when specifying the `format` option, see the [Python datetime documentation](#).

### 19.3.2 Assembling Datetime from Multiple DataFrame Columns

New in version 0.18.1.

You can also pass a `DataFrame` of integer or string columns to assemble into a `Series` of `Timestamps`.

```
In [31]: df = pd.DataFrame({'year': [2015, 2016],
.....:                    'month': [2, 3],
.....:                    'day': [4, 5],
.....:                    'hour': [2, 3]})
.....:
```

```
In [32]: pd.to_datetime(df)
```

```
Out[32]:
0    2015-02-04 02:00:00
1    2016-03-05 03:00:00
dtype: datetime64[ns]
```

You can pass only the columns that you need to assemble.

```
In [33]: pd.to_datetime(df[['year', 'month', 'day']])
```

```
Out[33]:
0    2015-02-04
1    2016-03-05
dtype: datetime64[ns]
```

`pd.to_datetime` looks for standard designations of the datetime component in the column names, including:

- required: year, month, day
- optional: hour, minute, second, millisecond, microsecond, nanosecond

### 19.3.3 Invalid Data

The default behavior, `errors='raise'`, is to raise when unparseable:

```
In [2]: pd.to_datetime(['2009/07/31', 'asd'], errors='raise')
ValueError: Unknown string format
```

Pass `errors='ignore'` to return the original input when unparseable:

```
In [34]: pd.to_datetime(['2009/07/31', 'asd'], errors='ignore')
Out[34]: array(['2009/07/31', 'asd'], dtype=object)
```

Pass `errors='coerce'` to convert unparseable data to `NaT` (not a time):

```
In [35]: pd.to_datetime(['2009/07/31', 'asd'], errors='coerce')
Out[35]: DatetimeIndex(['2009-07-31', 'NaT'], dtype='datetime64[ns]', freq=None)
```

### 19.3.4 Epoch Timestamps

pandas supports converting integer or float epoch times to `Timestamp` and `DatetimeIndex`. The default unit is nanoseconds, since that is how `Timestamp` objects are stored internally. However, epochs are often stored in another unit which can be specified. These are computed from the starting point specified by the `origin` parameter.

```
In [36]: pd.to_datetime([1349720105, 1349806505, 1349892905,
.....:                  1349979305, 1350065705], unit='s')
.....:

Out[36]:
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
              '2012-10-10 18:15:05', '2012-10-11 18:15:05',
              '2012-10-12 18:15:05'],
              dtype='datetime64[ns]', freq=None)

In [37]: pd.to_datetime([1349720105100, 1349720105200, 1349720105300,
.....:                  1349720105400, 1349720105500 ], unit='ms')
.....:

#####
↪
DatetimeIndex(['2012-10-08 18:15:05.100000', '2012-10-08 18:15:05.200000',
              '2012-10-08 18:15:05.300000', '2012-10-08 18:15:05.400000',
              '2012-10-08 18:15:05.500000'],
              dtype='datetime64[ns]', freq=None)
```

**Note:** Epoch times will be rounded to the nearest nanosecond.

**Warning:** Conversion of float epoch times can lead to inaccurate and unexpected results. `Python floats` have about 15 digits precision in decimal. Rounding during conversion from float to high precision `Timestamp` is unavoidable. The only way to achieve exact precision is to use a fixed-width types (e.g. an `int64`).

[illegible]

**See also:**

### Using the origin Parameter

### 19.3.5 From Timestamps to Epoch

To invert the operation from above, namely, to convert from a `Timestamp` to a ‘unix’ epoch:

```
In [40]: stamps = pd.date_range('2012-10-08 18:15:05', periods=4, freq='D')

In [41]: stamps
Out[41]:
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
              '2012-10-10 18:15:05', '2012-10-11 18:15:05'],
              dtype='datetime64[ns]', freq='D')
```

We subtract the epoch (midnight at January 1, 1970 UTC) and then floor divide by the “unit” (1 second).

```
In [42]: (stamps - pd.Timestamp("1970-01-01")) // pd.Timedelta('1s')
Out[42]: Int64Index([1349720105, 1349806505, 1349892905, 1349979305], dtype='int64')
```



### 19.3.6 Using the `origin` Parameter

New in version 0.20.0.

Using the `origin` parameter, one can specify an alternative starting point for creation of a `DatetimeIndex`. For example, to use 1960-01-01 as the starting date:

```
In [43]: pd.to_datetime([1, 2, 3], unit='D', origin=pd.Timestamp('1960-01-01'))
Out[43]: DatetimeIndex(['1960-01-02', '1960-01-03', '1960-01-04'], dtype=
↳ 'datetime64[ns]', freq=None)
```

The default is set at `origin='unix'`, which defaults to 1970-01-01 00:00:00. Commonly called ‘unix epoch’ or POSIX time.

```
In [44]: pd.to_datetime([1, 2, 3], unit='D')
Out[44]: DatetimeIndex(['1970-01-02', '1970-01-03', '1970-01-04'], dtype=
↳ 'datetime64[ns]', freq=None)
```

## 19.4 Generating Ranges of Timestamps

To generate an index with timestamps, you can use either the `DatetimeIndex` or `Index` constructor and pass in a list of datetime objects:

```
In [45]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]

# Note the frequency information
In [46]: index = pd.DatetimeIndex(dates)

In [47]: index
Out[47]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype=
↳ 'datetime64[ns]', freq=None)

# Automatically converted to DatetimeIndex
In [48]: index = pd.Index(dates)

In [49]: index
Out[49]: DatetimeIndex(['2012-05-01', '2012-05-02', '2012-05-03'], dtype=
↳ 'datetime64[ns]', freq=None)
```

In practice this becomes very cumbersome because we often need a very long index with a large number of timestamps. If we need timestamps on a regular frequency, we can use the `date_range()` and `bdate_range()` functions to create a `DatetimeIndex`. The default frequency for `date_range` is a **calendar day** while the default for `bdate_range` is a **business day**:

```
In [50]: start = datetime(2011, 1, 1)

In [51]: end = datetime(2012, 1, 1)

In [52]: index = pd.date_range(start, end)

In [53]: index
Out[53]:
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-01-04',
                '2011-01-05', '2011-01-06', '2011-01-07', '2011-01-08',
                '2011-01-09', '2011-01-10',
```

(continues on next page)

(continued from previous page)

```

...
'2011-12-23', '2011-12-24', '2011-12-25', '2011-12-26',
'2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30',
'2011-12-31', '2012-01-01'],
dtype='datetime64[ns]', length=366, freq='D')

In [54]: index = pd.bdate_range(start, end)

In [55]: index
Out[55]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14',
               ...
               '2011-12-19', '2011-12-20', '2011-12-21', '2011-12-22',
               '2011-12-23', '2011-12-26', '2011-12-27', '2011-12-28',
               '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', length=260, freq='B')

```

Convenience functions like `date_range` and `bdate_range` can utilize a variety of *frequency aliases*:

```

In [56]: pd.date_range(start, periods=1000, freq='M')
Out[56]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-30',
               '2011-05-31', '2011-06-30', '2011-07-31', '2011-08-31',
               '2011-09-30', '2011-10-31',
               ...
               '2093-07-31', '2093-08-31', '2093-09-30', '2093-10-31',
               '2093-11-30', '2093-12-31', '2094-01-31', '2094-02-28',
               '2094-03-31', '2094-04-30'],
              dtype='datetime64[ns]', length=1000, freq='M')

In [57]: pd.bdate_range(start, periods=250, freq='BQS')
↪
DatetimeIndex(['2011-01-03', '2011-04-01', '2011-07-01', '2011-10-03',
               '2012-01-02', '2012-04-02', '2012-07-02', '2012-10-01',
               '2013-01-01', '2013-04-01',
               ...
               '2071-01-01', '2071-04-01', '2071-07-01', '2071-10-01',
               '2072-01-01', '2072-04-01', '2072-07-01', '2072-10-03',
               '2073-01-02', '2073-04-03'],
              dtype='datetime64[ns]', length=250, freq='BQS-JAN')

```

`date_range` and `bdate_range` make it easy to generate a range of dates using various combinations of parameters like `start`, `end`, `periods`, and `freq`. The start and end dates are strictly inclusive, so dates outside of those specified will not be generated:

```

In [58]: pd.date_range(start, end, freq='BM')
Out[58]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [59]: pd.date_range(start, end, freq='W')

```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
DatetimeIndex(['2011-01-02', '2011-01-09', '2011-01-16', '2011-01-23',
               '2011-01-30', '2011-02-06', '2011-02-13', '2011-02-20',
               '2011-02-27', '2011-03-06', '2011-03-13', '2011-03-20',
               '2011-03-27', '2011-04-03', '2011-04-10', '2011-04-17',
               '2011-04-24', '2011-05-01', '2011-05-08', '2011-05-15',
               '2011-05-22', '2011-05-29', '2011-06-05', '2011-06-12',
               '2011-06-19', '2011-06-26', '2011-07-03', '2011-07-10',
               '2011-07-17', '2011-07-24', '2011-07-31', '2011-08-07',
               '2011-08-14', '2011-08-21', '2011-08-28', '2011-09-04',
               '2011-09-11', '2011-09-18', '2011-09-25', '2011-10-02',
               '2011-10-09', '2011-10-16', '2011-10-23', '2011-10-30',
               '2011-11-06', '2011-11-13', '2011-11-20', '2011-11-27',
               '2011-12-04', '2011-12-11', '2011-12-18', '2011-12-25',
               '2012-01-01'],
              dtype='datetime64[ns]', freq='W-SUN')

```

```
In [60]: pd.bdate_range(end=end, periods=20)
```

```

////////////////////////////////////
↪
DatetimeIndex(['2011-12-05', '2011-12-06', '2011-12-07', '2011-12-08',
               '2011-12-09', '2011-12-12', '2011-12-13', '2011-12-14',
               '2011-12-15', '2011-12-16', '2011-12-19', '2011-12-20',
               '2011-12-21', '2011-12-22', '2011-12-23', '2011-12-26',
               '2011-12-27', '2011-12-28', '2011-12-29', '2011-12-30'],
              dtype='datetime64[ns]', freq='B')

```

```
In [61]: pd.bdate_range(start=start, periods=20)
```

```

////////////////////////////////////
↪
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07', '2011-01-10', '2011-01-11', '2011-01-12',
               '2011-01-13', '2011-01-14', '2011-01-17', '2011-01-18',
               '2011-01-19', '2011-01-20', '2011-01-21', '2011-01-24',
               '2011-01-25', '2011-01-26', '2011-01-27', '2011-01-28'],
              dtype='datetime64[ns]', freq='B')

```

New in version 0.23.0.

Specifying start, end, and periods will generate a range of evenly spaced dates from start to end inclusively, with periods number of elements in the resulting DatetimeIndex:

```
In [62]: pd.date_range('2018-01-01', '2018-01-05', periods=5)
```

```
Out [62]:
```

```

DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05'],
              dtype='datetime64[ns]', freq=None)

```

```
In [63]: pd.date_range('2018-01-01', '2018-01-05', periods=10)
```

```

////////////////////////////////////
↪
DatetimeIndex(['2018-01-01 00:00:00', '2018-01-01 10:40:00',
               '2018-01-01 21:20:00', '2018-01-02 08:00:00',
               '2018-01-02 18:40:00', '2018-01-03 05:20:00',
               '2018-01-03 16:00:00', '2018-01-04 02:40:00',
               '2018-01-04 13:20:00', '2018-01-05 00:00:00'],
              dtype='datetime64[ns]', freq=None)

```

(continues on next page)

(continued from previous page)

```
dtype='datetime64[ns]', freq=None)
```

## 19.4.1 Custom Frequency Ranges

**Warning:** This functionality was originally exclusive to `cdate_range`, which is deprecated as of version 0.21.0 in favor of `bdate_range`. Note that `cdate_range` only utilizes the `weekmask` and `holidays` parameters when custom business day, 'C', is passed as the frequency string. Support has been expanded with `bdate_range` to work with any custom frequency string.

New in version 0.21.0.

`bdate_range` can also generate a range of custom frequency dates by using the `weekmask` and `holidays` parameters. These parameters will only be used if a custom frequency string is passed.

```
In [64]: weekmask = 'Mon Wed Fri'

In [65]: holidays = [datetime(2011, 1, 5), datetime(2011, 3, 14)]

In [66]: pd.bdate_range(start, end, freq='C', weekmask=weekmask, holidays=holidays)
Out [66]:
DatetimeIndex(['2011-01-03', '2011-01-07', '2011-01-10', '2011-01-12',
               '2011-01-14', '2011-01-17', '2011-01-19', '2011-01-21',
               '2011-01-24', '2011-01-26',
               ...
               '2011-12-09', '2011-12-12', '2011-12-14', '2011-12-16',
               '2011-12-19', '2011-12-21', '2011-12-23', '2011-12-26',
               '2011-12-28', '2011-12-30'],
              dtype='datetime64[ns]', length=154, freq='C')

In [67]: pd.bdate_range(start, end, freq='CBMS', weekmask=weekmask)
Out [67]:
DatetimeIndex(['2011-01-03', '2011-02-02', '2011-03-02', '2011-04-01',
               '2011-05-02', '2011-06-01', '2011-07-01', '2011-08-01',
               '2011-09-02', '2011-10-03', '2011-11-02', '2011-12-02'],
              dtype='datetime64[ns]', freq='CBMS')
```

See also:

*Custom Business Days*

## 19.5 Timestamp Limitations

Since pandas represents timestamps in nanosecond resolution, the time span that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [68]: pd.Timestamp.min
Out [68]: Timestamp('1677-09-21 00:12:43.145225')

In [69]: pd.Timestamp.max
Out [69]: Timestamp('2262-04-11 23:47:16.854775807')
```

(continues on next page)

(continued from previous page)

**See also:***Representing Out-of-Bounds Spans*

## 19.6 Indexing

One of the main uses for `DatetimeIndex` is as an index for pandas objects. The `DatetimeIndex` class contains many time series related optimizations:

- A large range of dates for various offsets are pre-computed and cached under the hood in order to make generating subsequent date ranges very fast (just have to grab a slice).
- Fast shifting using the `shift` and `tshift` method on pandas objects.
- Unioning of overlapping `DatetimeIndex` objects with the same frequency is very fast (important for fast data alignment).
- Quick access to date fields via properties such as `year`, `month`, etc.
- Regularization functions like `snap` and very fast `asof` logic.

`DatetimeIndex` objects have all the basic functionality of regular `Index` objects, and a smorgasbord of advanced time series specific methods for easy frequency processing.

**See also:***Reindexing methods*

**Note:** While pandas does not force you to have a sorted date index, some of these methods may have unexpected or incorrect behavior if the dates are unsorted.

`DatetimeIndex` can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

```
In [70]: rng = pd.date_range(start, end, freq='BM')

In [71]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [72]: ts.index
Out[72]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31', '2011-06-30', '2011-07-29', '2011-08-31',
               '2011-09-30', '2011-10-31', '2011-11-30', '2011-12-30'],
              dtype='datetime64[ns]', freq='BM')

In [73]: ts[:5].index
Out[73]:
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-04-29',
               '2011-05-31'],
              dtype='datetime64[ns]', freq='BM')

In [74]: ts[::2].index
Out[74]:
DatetimeIndex(['2011-01-31', '2011-03-31', '2011-05-31', '2011-07-29',
               '2011-09-30', '2011-11-30'],
              dtype='datetime64[ns]', freq='BMS')
```

(continues on next page)

(continued from previous page)

```
DatetimeIndex(['2011-01-31', '2011-03-31', '2011-05-31', '2011-07-29',
               '2011-09-30', '2011-11-30'],
              dtype='datetime64[ns]', freq='2BM')
```

## 19.6.1 Partial String Indexing

Dates and strings that parse to timestamps can be passed as indexing parameters:

```
In [75]: ts['1/31/2011']
Out[75]: -1.2812473076599531

In [76]: ts[datetime(2011, 12, 25):]
Out[76]:
2011-12-30    0.687738
Freq: BM, dtype: float64

In [77]: ts['10/31/2011':'12/31/2011']
Out[77]:
2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-30    0.687738
Freq: BM, dtype: float64
```

To provide convenience for accessing longer time series, you can also pass in the year or year and month as strings:

```
In [78]: ts['2011']
Out[78]:
2011-01-31   -1.281247
2011-02-28   -0.727707
2011-03-31   -0.121306
2011-04-29   -0.097883
2011-05-31    0.695775
2011-06-30    0.341734
2011-07-29    0.959726
2011-08-31   -1.110336
2011-09-30   -0.619976
2011-10-31    0.149748
2011-11-30   -0.732339
2011-12-30    0.687738
Freq: BM, dtype: float64

In [79]: ts['2011-6']
Out[79]:
2011-06-30    0.341734
Freq: BM, dtype: float64
```

This type of slicing will work on a `DataFrame` with a `DatetimeIndex` as well. Since the partial string selection is a form of label slicing, the endpoints **will be** included. This would include matching times on an included date:

```
In [80]: dft = pd.DataFrame(randn(100000, 1),
.....:                      columns=['A'],
.....:                      index=pd.date_range('20130101', periods=100000, freq='T'))
```

(continues on next page)

(continued from previous page)

**In [81]:** dft**Out [81]:**

```

          A
2013-01-01 00:00:00    0.176444
2013-01-01 00:01:00    0.403310
2013-01-01 00:02:00   -0.154951
2013-01-01 00:03:00    0.301624
2013-01-01 00:04:00   -2.179861
2013-01-01 00:05:00   -1.369849
2013-01-01 00:06:00   -0.954208
...
2013-03-11 10:33:00  -0.293083
2013-03-11 10:34:00  -0.059881
2013-03-11 10:35:00    1.252450
2013-03-11 10:36:00    0.046611
2013-03-11 10:37:00    0.059478
2013-03-11 10:38:00  -0.286539
2013-03-11 10:39:00    0.841669

```

[100000 rows x 1 columns]

**In [82]:** dft['2013']

```

////////////////////////////////////
↪

```

```

          A
2013-01-01 00:00:00    0.176444
2013-01-01 00:01:00    0.403310
2013-01-01 00:02:00   -0.154951
2013-01-01 00:03:00    0.301624
2013-01-01 00:04:00   -2.179861
2013-01-01 00:05:00   -1.369849
2013-01-01 00:06:00   -0.954208
...
2013-03-11 10:33:00  -0.293083
2013-03-11 10:34:00  -0.059881
2013-03-11 10:35:00    1.252450
2013-03-11 10:36:00    0.046611
2013-03-11 10:37:00    0.059478
2013-03-11 10:38:00  -0.286539
2013-03-11 10:39:00    0.841669

```

[100000 rows x 1 columns]

This starts on the very first time in the month, and includes the last date and time for the month:

**In [83]:** dft['2013-1':'2013-2']**Out [83]:**

```

          A
2013-01-01 00:00:00    0.176444
2013-01-01 00:01:00    0.403310
2013-01-01 00:02:00   -0.154951
2013-01-01 00:03:00    0.301624
2013-01-01 00:04:00   -2.179861
2013-01-01 00:05:00   -1.369849
2013-01-01 00:06:00   -0.954208
...

```

(continues on next page)

(continued from previous page)

```
2013-02-28 23:53:00    0.103114
2013-02-28 23:54:00   -1.303422
2013-02-28 23:55:00    0.451943
2013-02-28 23:56:00    0.220534
2013-02-28 23:57:00   -1.624220
2013-02-28 23:58:00    0.093915
2013-02-28 23:59:00   -1.087454

[84960 rows x 1 columns]
```

This specifies a stop time **that includes all of the times on the last day**:

```
In [84]: dft['2013-1':'2013-2-28']
Out[84]:
```

|                     | A         |
|---------------------|-----------|
| 2013-01-01 00:00:00 | 0.176444  |
| 2013-01-01 00:01:00 | 0.403310  |
| 2013-01-01 00:02:00 | -0.154951 |
| 2013-01-01 00:03:00 | 0.301624  |
| 2013-01-01 00:04:00 | -2.179861 |
| 2013-01-01 00:05:00 | -1.369849 |
| 2013-01-01 00:06:00 | -0.954208 |
| ...                 | ...       |
| 2013-02-28 23:53:00 | 0.103114  |
| 2013-02-28 23:54:00 | -1.303422 |
| 2013-02-28 23:55:00 | 0.451943  |
| 2013-02-28 23:56:00 | 0.220534  |
| 2013-02-28 23:57:00 | -1.624220 |
| 2013-02-28 23:58:00 | 0.093915  |
| 2013-02-28 23:59:00 | -1.087454 |

```
[84960 rows x 1 columns]
```

This specifies an **exact** stop time (and is not the same as the above):

```
In [85]: dft['2013-1':'2013-2-28 00:00:00']
Out[85]:
```

|                     | A         |
|---------------------|-----------|
| 2013-01-01 00:00:00 | 0.176444  |
| 2013-01-01 00:01:00 | 0.403310  |
| 2013-01-01 00:02:00 | -0.154951 |
| 2013-01-01 00:03:00 | 0.301624  |
| 2013-01-01 00:04:00 | -2.179861 |
| 2013-01-01 00:05:00 | -1.369849 |
| 2013-01-01 00:06:00 | -0.954208 |
| ...                 | ...       |
| 2013-02-27 23:54:00 | 0.897051  |
| 2013-02-27 23:55:00 | -0.309230 |
| 2013-02-27 23:56:00 | 1.944713  |
| 2013-02-27 23:57:00 | 0.369265  |
| 2013-02-27 23:58:00 | 0.053071  |
| 2013-02-27 23:59:00 | -0.019734 |
| 2013-02-28 00:00:00 | 1.388189  |

```
[83521 rows x 1 columns]
```

We are stopping on the included end-point as it is part of the index:



```
In [86]: dft['2013-1-15':'2013-1-15 12:30:00']
Out[86]:
```

```

          A
2013-01-15 00:00:00  0.501288
2013-01-15 00:01:00 -0.605198
2013-01-15 00:02:00  0.215146
2013-01-15 00:03:00  0.924732
2013-01-15 00:04:00 -2.228519
2013-01-15 00:05:00  1.517331
2013-01-15 00:06:00 -1.188774
...
2013-01-15 12:24:00  1.358314
2013-01-15 12:25:00 -0.737727
2013-01-15 12:26:00  1.838323
2013-01-15 12:27:00 -0.774090
2013-01-15 12:28:00  0.622261
2013-01-15 12:29:00 -0.631649
2013-01-15 12:30:00  0.193284

[751 rows x 1 columns]
```

New in version 0.18.0.

DatetimeIndex partial string indexing also works on a DataFrame with a MultiIndex:

```
In [87]: dft2 = pd.DataFrame(np.random.randn(20, 1),
    ....:                    columns=['A'],
    ....:                    index=pd.MultiIndex.from_product([pd.date_range('20130101',
↳ ',
    ....:
↳ periods=10,
    ....:
    ....:                    freq='12H
↳ '),
    ....:                    ['a', 'b']]))
    ....:
```

```
In [88]: dft2
Out[88]:
```

```

          A
2013-01-01 00:00:00 a -0.659574
                   b  1.494522
2013-01-01 12:00:00 a -0.778425
                   b -0.253355
2013-01-02 00:00:00 a -2.816159
                   b -1.210929
2013-01-02 12:00:00 a  0.144669
...
2013-01-04 00:00:00 b -1.624463
2013-01-04 12:00:00 a  0.056912
                   b  0.149867
2013-01-05 00:00:00 a -1.256173
                   b  2.324544
2013-01-05 12:00:00 a -1.067396
                   b -0.660996

[20 rows x 1 columns]
```

```
In [89]: dft2.loc['2013-01-05']
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
                                     A
2013-01-05 00:00:00 a -1.256173
                                     b  2.324544
2013-01-05 12:00:00 a -1.067396
                                     b -0.660996

In [90]: idx = pd.IndexSlice

In [91]: dft2 = dft2.swaplevel(0, 1).sort_index()

In [92]: dft2.loc[idx[:, '2013-01-05'], :]
Out[92]:
                                     A
a 2013-01-05 00:00:00 -1.256173
   2013-01-05 12:00:00 -1.067396
b 2013-01-05 00:00:00  2.324544
   2013-01-05 12:00:00 -0.660996

```

## 19.6.2 Slice vs. Exact Match

Changed in version 0.20.0.

The same string used as an indexing parameter can be treated either as a slice or as an exact match depending on the resolution of the index. If the string is less accurate than the index, it will be treated as a slice, otherwise as an exact match.

Consider a `Series` object with a minute resolution index:

```

In [93]: series_minute = pd.Series([1, 2, 3],
.....:                             pd.DatetimeIndex(['2011-12-31 23:59:00',
.....:                                                  '2012-01-01 00:00:00',
.....:                                                  '2012-01-01 00:02:00']))
.....:

In [94]: series_minute.index.resolution
Out[94]: 'minute'

```

A timestamp string less accurate than a minute gives a `Series` object.

```

In [95]: series_minute['2011-12-31 23']
Out[95]:
2011-12-31 23:59:00    1
dtype: int64

```

A timestamp string with minute resolution (or more accurate), gives a scalar instead, i.e. it is not casted to a slice.

```

In [96]: series_minute['2011-12-31 23:59']
Out[96]: 1

In [97]: series_minute['2011-12-31 23:59:00']
Out[97]: 1

```

If index resolution is second, then the minute-accurate timestamp gives a `Series`.

```
In [98]: series_second = pd.Series([1, 2, 3],
.....:                             pd.DatetimeIndex(['2011-12-31 23:59:59',
.....:                                                  '2012-01-01 00:00:00',
.....:                                                  '2012-01-01 00:00:01']))
.....:

In [99]: series_second.index.resolution
Out[99]: 'second'

In [100]: series_second['2011-12-31 23:59']
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[100]:
2011-12-31 23:59:59      1
dtype: int64
```

If the timestamp string is treated as a slice, it can be used to index DataFrame with `[]` as well.

```
In [101]: dft_minute = pd.DataFrame({'a': [1, 2, 3], 'b': [4, 5, 6]},
.....:                               index=series_minute.index)
.....:

In [102]: dft_minute['2011-12-31 23']
Out[102]:
```

|                     | a | b |
|---------------------|---|---|
| 2011-12-31 23:59:00 | 1 | 4 |

**Warning:** However, if the string is treated as an exact match, the selection in DataFrame's `[]` will be column-wise and not row-wise, see [Indexing Basics](#). For example `dft_minute['2011-12-31 23:59']` will raise `KeyError` as '2012-12-31 23:59' has the same resolution as the index and there is no column with such name:

To *always* have unambiguous selection, whether the row is treated as a slice or a single selection, use `.loc`.

```
In [103]: dft_minute.loc['2011-12-31 23:59']
Out[103]:
```

|                     | a | b |
|---------------------|---|---|
| 2011-12-31 23:59:00 | 1 | 4 |

Name: 2011-12-31 23:59:00, dtype: int64

Note also that `DatetimeIndex` resolution cannot be less precise than day.

```
In [104]: series_monthly = pd.Series([1, 2, 3],
.....:                               pd.DatetimeIndex(['2011-12',
.....:                                                  '2012-01',
.....:                                                  '2012-02']))
.....:

In [105]: series_monthly.index.resolution
Out[105]: 'day'

In [106]: series_monthly['2011-12'] # returns Series
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[106]:
2011-12-01      1
dtype: int64
```

### 19.6.3 Exact Indexing

As discussed in previous section, indexing a `DatetimeIndex` with a partial string depends on the “accuracy” of the period, in other words how specific the interval is in relation to the resolution of the index. In contrast, indexing with `Timestamp` or `datetime` objects is exact, because the objects have exact meaning. These also follow the semantics of *including both endpoints*.

These `Timestamp` and `datetime` objects have exact hours, minutes, and seconds, even though they were not explicitly specified (they are 0).

```
In [107]: dft[datetime(2013, 1, 1):datetime(2013,2,28)]
Out[107]:
```

|                     | A         |
|---------------------|-----------|
| 2013-01-01 00:00:00 | 0.176444  |
| 2013-01-01 00:01:00 | 0.403310  |
| 2013-01-01 00:02:00 | -0.154951 |
| 2013-01-01 00:03:00 | 0.301624  |
| 2013-01-01 00:04:00 | -2.179861 |
| 2013-01-01 00:05:00 | -1.369849 |
| 2013-01-01 00:06:00 | -0.954208 |
| ...                 | ...       |
| 2013-02-27 23:54:00 | 0.897051  |
| 2013-02-27 23:55:00 | -0.309230 |
| 2013-02-27 23:56:00 | 1.944713  |
| 2013-02-27 23:57:00 | 0.369265  |
| 2013-02-27 23:58:00 | 0.053071  |
| 2013-02-27 23:59:00 | -0.019734 |
| 2013-02-28 00:00:00 | 1.388189  |

[83521 rows x 1 columns]

With no defaults.

```
In [108]: dft[datetime(2013, 1, 1, 10, 12, 0):datetime(2013, 2, 28, 10, 12, 0)]
Out[108]:
```

|                     | A         |
|---------------------|-----------|
| 2013-01-01 10:12:00 | -0.246733 |
| 2013-01-01 10:13:00 | -1.429225 |
| 2013-01-01 10:14:00 | -1.265339 |
| 2013-01-01 10:15:00 | 0.710986  |
| 2013-01-01 10:16:00 | -0.818200 |
| 2013-01-01 10:17:00 | 0.543542  |
| 2013-01-01 10:18:00 | 1.577713  |
| ...                 | ...       |
| 2013-02-28 10:06:00 | 0.311249  |
| 2013-02-28 10:07:00 | 2.366080  |
| 2013-02-28 10:08:00 | -0.490372 |
| 2013-02-28 10:09:00 | 0.373340  |
| 2013-02-28 10:10:00 | 0.638442  |
| 2013-02-28 10:11:00 | 1.330135  |
| 2013-02-28 10:12:00 | -0.945450 |

[83521 rows x 1 columns]

## 19.6.4 Truncating & Fancy Indexing

A `truncate()` convenience function is provided that is similar to slicing. Note that `truncate` assumes a 0 value for any unspecified date component in a `DatetimeIndex` in contrast to slicing which returns any partially matching dates:

```
In [109]: rng2 = pd.date_range('2011-01-01', '2012-01-01', freq='W')
```

```
In [110]: ts2 = pd.Series(np.random.randn(len(rng2)), index=rng2)
```

```
In [111]: ts2.truncate(before='2011-11', after='2011-12')
```

```
Out[111]:
```

```
2011-11-06    -0.773743
2011-11-13     0.247216
2011-11-20     0.591308
2011-11-27     2.228500
Freq: W-SUN, dtype: float64
```

```
In [112]: ts2['2011-11':'2011-12']
```

```
////////////////////////////////////
```

```
↪
2011-11-06    -0.773743
2011-11-13     0.247216
2011-11-20     0.591308
2011-11-27     2.228500
2011-12-04     0.838769
2011-12-11     0.658538
2011-12-18     0.567353
2011-12-25    -1.076735
Freq: W-SUN, dtype: float64
```

Even complicated fancy indexing that breaks the `DatetimeIndex` frequency regularity will result in a `DatetimeIndex`, although frequency is lost:

```
In [113]: ts2[[0, 2, 6]].index
```

```
Out[113]: DatetimeIndex(['2011-01-02', '2011-01-16', '2011-02-13'], dtype=
↪ 'datetime64[ns]', freq=None)
```

## 19.7 Time/Date Components

There are several time/date properties that one can access from `Timestamp` or a collection of timestamps like a `DatetimeIndex`.

| Property         | Description   |
|------------------|---|
| year             | The year of the datetime  |
| month            | The month of the datetime   |
| day              | The days of the datetime  |
| hour             | The hour of the datetime  |
| minute           | The minutes of the datetime                                       |
| second           | The seconds of the datetime                                       |
| microsecond      | The microseconds of the datetime                                  |
| nanosecond       | The nanoseconds of the datetime                                   |
| date             | Returns datetime.date (does not contain timezone information)     |
| time             | Returns datetime.time (does not contain timezone information)     |
| dayofyear        | The ordinal day of year   |
| weekofyear       | The week ordinal of the year                                      |
| week             | The week ordinal of the year                                      |
| dayofweek        | The number of the day of the week with Monday=0, Sunday=6         |
| weekday          | The number of the day of the week with Monday=0, Sunday=6         |
| weekday_name     | The name of the day in a week (ex: Friday)                        |
| quarter          | Quarter of the date: Jan-Mar = 1, Apr-Jun = 2, etc.               |
| days_in_month    | The number of days in the month of the datetime                   |
| is_month_start   | Logical indicating if first day of month (defined by frequency)   |
| is_month_end     | Logical indicating if last day of month (defined by frequency)    |
| is_quarter_start | Logical indicating if first day of quarter (defined by frequency) |
| is_quarter_end   | Logical indicating if last day of quarter (defined by frequency)  |
| is_year_start    | Logical indicating if first day of year (defined by frequency)    |
| is_year_end      | Logical indicating if last day of year (defined by frequency)     |
| is_leap_year     | Logical indicating if the date belongs to a leap year             |

Furthermore, if you have a `Series` with datetimelike values, then you can access these properties via the `.dt` accessor, as detailed in the section on [.dt accessors](#).

## 19.8 DateOffset Objects

In the preceding examples, we created `DatetimeIndex` objects at various frequencies by passing in *frequency strings* like ‘M’, ‘W’, and ‘BM’ to the `freq` keyword. Under the hood, these frequency strings are being translated into an instance of `DateOffset`, which represents a regular frequency increment. Specific offset logic like “month”, “business day”, or “one hour” is represented in its various subclasses.

| Class name                   | Description  |
|------------------------------|--|
| <code>DateOffset</code>      | Generic offset class, defaults to 1 calendar day   |
| <code>BDay</code>            | business day (weekday)                             |
| <code>CDay</code>            | custom business day                                |
| <code>Week</code>            | one week, optionally anchored on a day of the week |
| <code>WeekOfMonth</code>     | the x-th day of the y-th week of each month        |
| <code>LastWeekOfMonth</code> | the x-th day of the last week of each month        |
| <code>MonthEnd</code>        | calendar month end                                 |
| <code>MonthBegin</code>      | calendar month begin                               |
| <code>BMonthEnd</code>       | business month end                                 |
| <code>BMonthBegin</code>     | business month begin                               |
| <code>CBMonthEnd</code>      | custom business month end                          |

Continued on next page

Table 1 – continued from previous page

| Class name         | Description   |
|--------------------|---|
| CBMonthBegin       | custom business month begin                           |
| SemiMonthEnd       | 15th (or other day_of_month) and calendar month end   |
| SemiMonthBegin     | 15th (or other day_of_month) and calendar month begin |
| QuarterEnd         | calendar quarter end                                  |
| QuarterBegin       | calendar quarter begin                                |
| BQuarterEnd        | business quarter end                                  |
| BQuarterBegin      | business quarter begin                                |
| FY5253Quarter      | retail (aka 52-53 week) quarter                       |
| YearEnd            | calendar year end                                     |
| YearBegin          | calendar year begin                                   |
| BYearEnd           | business year end                                     |
| BYearBegin         | business year begin                                   |
| FY5253             | retail (aka 52-53 week) year                          |
| BusinessHour       | business hour   |
| CustomBusinessHour | custom business hour                                  |
| Hour               | one hour  |
| Minute             | one minute  |
| Second             | one second  |
| Milli              | one millisecond                                       |
| Micro              | one microsecond                                       |
| Nano               | one nanosecond  |

The basic `DateOffset` takes the same arguments as `dateutil.relativedelta`, which works as follows:

```
In [114]: d = datetime(2008, 8, 18, 9, 0)

In [115]: d + relativedelta(months=4, days=5)
Out[115]: datetime.datetime(2008, 12, 23, 9, 0)
```

We could have done the same thing with `DateOffset`:

```
In [116]: from pandas.tseries.offsets import *

In [117]: d + DateOffset(months=4, days=5)
Out[117]: Timestamp('2008-12-23 09:00:00')
```

The key features of a `DateOffset` object are:

- It can be added / subtracted to/from a datetime object to obtain a shifted date.
- It can be multiplied by an integer (positive or negative) so that the increment will be applied multiple times.
- It has `rollforward()` and `rollback()` methods for moving a date forward or backward to the next or previous “offset date”.

Subclasses of `DateOffset` define the `apply` function which dictates custom date increment logic, such as adding business days:

```
class BDay(DateOffset):
    """DateOffset increments between business days"""
    def apply(self, other):
        ...
```

```
In [118]: d - 5 * BDay()
Out[118]: Timestamp('2008-08-11 09:00:00')

In [119]: d + BMonthEnd()
Out[119]: Timestamp('2008-08-29 09:00:00')
```

The `rollforward` and `rollback` methods do exactly what you would expect:

[illegible]

It's definitely worth exploring the `pandas.tseries.offsets` module and the various docstrings for the classes.

These operations (apply, rollforward and rollback) preserve time (hour, minute, etc) information by default. To reset time, use `normalize=True` when creating the offset instance. If `normalize=True`, the result is normalized after the function is applied.

```
In [124]: day = Day()

In [125]: day.apply(pd.Timestamp('2014-01-01 09:00'))
Out[125]: Timestamp('2014-01-02 09:00:00')

In [126]: day = Day(normalize=True)

In [127]: day.apply(pd.Timestamp('2014-01-01 09:00'))
Out[127]: Timestamp('2014-01-02 00:00:00')

In [128]: hour = Hour()

In [129]: hour.apply(pd.Timestamp('2014-01-01 22:00'))
Out[129]: Timestamp('2014-01-01 23:00:00')

In [130]: hour = Hour(normalize=True)

In [131]: hour.apply(pd.Timestamp('2014-01-01 22:00'))
Out[131]: Timestamp('2014-01-01 00:00:00')

In [132]: hour.apply(pd.Timestamp('2014-01-01 23:00'))
Out[132]: Timestamp('2014-01-02 00:00:00')
```

### 19.8.1 Parametric Offsets

Some of the offsets can be “parameterized” when created to result in different behaviors. For example, the `Week` offset for generating weekly data accepts a `weekday` parameter which results in the generated dates always lying on a particular day of the week:

```
In [133]: d
Out[133]: datetime.datetime(2008, 8, 18, 9, 0)
```

(continues on next page)



(continued from previous page)

```

In [134]: d + Week()
Out[134]: Timestamp('2008-08-25 09:00:00')

In [135]: d + Week(weekday=4)
Out[135]: Timestamp('2008-08-22 09:00:00')

In [136]: (d + Week(weekday=4)).weekday()
Out[136]: 4

In [137]: d - Week()
Out[137]: Timestamp('2008-08-11 09:00:00')

```

The `normalize` option will be effective for addition and subtraction.

```

In [138]: d + Week(normalize=True)
Out[138]: Timestamp('2008-08-25 00:00:00')

In [139]: d - Week(normalize=True)
Out[139]: Timestamp('2008-08-11 00:00:00')

```

Another example is parameterizing `YearEnd` with the specific ending month:

```

In [140]: d + YearEnd()
Out[140]: Timestamp('2008-12-31 09:00:00')

In [141]: d + YearEnd(month=6)
Out[141]: Timestamp('2009-06-30 09:00:00')

```

## 19.8.2 Using Offsets with Series / DatetimeIndex

Offsets can be used with either a `Series` or `DatetimeIndex` to apply the offset to each element.

```

In [142]: rng = pd.date_range('2012-01-01', '2012-01-03')

In [143]: s = pd.Series(rng)

In [144]: rng
Out[144]: DatetimeIndex(['2012-01-01', '2012-01-02', '2012-01-03'], dtype='datetime64[ns]', freq='D')

In [145]: rng + DateOffset(months=2)
Out[145]: DatetimeIndex(['2012-03-01', '2012-03-02', '2012-03-03'], dtype='datetime64[ns]', freq='D')

In [146]: s + DateOffset(months=2)
Out[146]:
0    2012-03-01
1    2012-03-02

```

(continues on next page)

(continued from previous page)

```
2    2012-03-03
dtype: datetime64[ns]

In [147]: s - DateOffset(months=2)
//////////
0    2011-11-01
1    2011-11-02
2    2011-11-03
dtype: datetime64[ns]
```

If the offset class maps directly to a `Timedelta` (Day, Hour, Minute, Second, Micro, Milli, Nano) it can be used exactly like a `Timedelta` - see the [Timedelta section](#) for more examples.

```
In [148]: s - Day(2)
Out[148]:
0    2011-12-30
1    2011-12-31
2    2012-01-01
dtype: datetime64[ns]

In [149]: td = s - pd.Series(pd.date_range('2011-12-29', '2011-12-31'))

In [150]: td
Out[150]:
0    3 days
1    3 days
2    3 days
dtype: timedelta64[ns]

In [151]: td + Minute(15)
Out[151]:
0    3 days 00:15:00
1    3 days 00:15:00
2    3 days 00:15:00
dtype: timedelta64[ns]
```

Note that some offsets (such as `BQuarterEnd`) do not have a vectorized implementation. They can still be used but may calculate significantly slower and will show a `PerformanceWarning`

```
In [152]: rng + BQuarterEnd()
Out[152]: DatetimeIndex(['2012-03-30', '2012-03-30', '2012-03-30'], dtype=
↳ 'datetime64[ns]', freq='D')
```

### 19.8.3 Custom Business Days

The `CDay` or `CustomBusinessDay` class provides a parametric `BusinessDay` class which can be used to create customized business day calendars which account for local holidays and local weekend conventions.

As an interesting example, let's look at Egypt where a Friday-Saturday weekend is observed.

```
In [153]: from pandas.tseries.offsets import CustomBusinessDay

In [154]: weekmask_egypt = 'Sun Mon Tue Wed Thu'
```

(continues on next page)

(continued from previous page)

```
# They also observe International Workers' Day so let's
# add that for a couple of years
In [155]: holidays = ['2012-05-01', datetime(2013, 5, 1), np.datetime64('2014-05-01')]

In [156]: bday_egypt = CustomBusinessDay(holidays=holidays, weekmask=weekmask_egypt)

In [157]: dt = datetime(2013, 4, 30)

In [158]: dt + 2 * bday_egypt
Out[158]: Timestamp('2013-05-05 00:00:00')
```

Let's map to the weekday names:

```
In [159]: dts = pd.date_range(dt, periods=5, freq=bday_egypt)

In [160]: pd.Series(dts.weekday, dts).map(pd.Series('Mon Tue Wed Thu Fri Sat Sun'.
↳split()))
Out[160]:
2013-04-30    Tue
2013-05-02    Thu
2013-05-05    Sun
2013-05-06    Mon
2013-05-07    Tue
Freq: C, dtype: object
```

Holiday calendars can be used to provide the list of holidays. See the [holiday calendar](#) section for more information.

```
In [161]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [162]: bday_us = CustomBusinessDay(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [163]: dt = datetime(2014, 1, 17)

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [164]: dt + bday_us
Out[164]: Timestamp('2014-01-21 00:00:00')
```

Monthly offsets that respect a certain holiday calendar can be defined in the usual way.

```
In [165]: from pandas.tseries.offsets import CustomBusinessMonthBegin

In [166]: bmth_us = CustomBusinessMonthBegin(calendar=USFederalHolidayCalendar())

# Skip new years
In [167]: dt = datetime(2013, 12, 17)

In [168]: dt + bmth_us
Out[168]: Timestamp('2014-01-02 00:00:00')

# Define date index with custom offset
In [169]: pd.DatetimeIndex(start='20100101', end='20120101', freq=bmth_us)
Out[169]:
DatetimeIndex(['2010-01-04', '2010-02-01', '2010-03-01', '2010-04-01',
                '2010-05-03', '2010-06-01', '2010-07-01', '2010-08-02',
                '2010-09-01', '2010-10-01', '2010-11-01', '2010-12-01',
                '2011-01-03', '2011-02-01', '2011-03-01', '2011-04-01',
```

(continues on next page)

(continued from previous page)

```
'2011-05-02', '2011-06-01', '2011-07-01', '2011-08-01',
'2011-09-01', '2011-10-03', '2011-11-01', '2011-12-01'],
dtype='datetime64[ns]', freq='CBMS')
```

**Note:** The frequency string ‘C’ is used to indicate that a CustomBusinessDay DateOffset is used, it is important to note that since CustomBusinessDay is a parameterised type, instances of CustomBusinessDay may differ and this is not detectable from the ‘C’ frequency string. The user therefore needs to ensure that the ‘C’ frequency string is used consistently within the user’s application.

## 19.8.4 Business Hour

The BusinessHour class provides a business hour representation on BusinessDay, allowing to use specific start and end times.

By default, BusinessHour uses 9:00 - 17:00 as business hours. Adding BusinessHour will increment Timestamp by hourly frequency. If target Timestamp is out of business hours, move to the next business hour then increment it. If the result exceeds the business hours end, the remaining hours are added to the next business day.

```
In [170]: bh = BusinessHour()

In [171]: bh
Out[171]: <BusinessHour: BH=09:00-17:00>

# 2014-08-01 is Friday
In [172]: pd.Timestamp('2014-08-01 10:00').weekday()
Out[172]: 4

In [173]: pd.Timestamp('2014-08-01 10:00') + bh
Out[173]: Timestamp('2014-08-01 11:00:00')

# Below example is the same as: pd.Timestamp('2014-08-01 09:00') + bh
In [174]: pd.Timestamp('2014-08-01 08:00') + bh
Out[174]: Timestamp('2014-08-01 10:00:00')

# If the results is on the end time, move to the next business day
In [175]: pd.Timestamp('2014-08-01 16:00') + bh
Out[175]: Timestamp('2014-08-04 09:00:00')

# Remainings are added to the next day
In [176]: pd.Timestamp('2014-08-01 16:30') + bh
Out[176]: Timestamp('2014-08-04 09:30:00')

# Adding 2 business hours
In [177]: pd.Timestamp('2014-08-01 10:00') + BusinessHour(2)
Out[177]: Timestamp('2014-08-01 12:00:00')

# Subtracting 3 business hours
```

(continues on next page)

(continued from previous page)

```
In [178]: pd.Timestamp('2014-08-01 10:00') + BusinessHour(-3)
\\Out[178]: Timestamp('2014-07-31 15:00:00')
```

You can also specify start and end time by keywords. The argument must be a `str` with an `hour:minute` representation or a `datetime.time` instance. Specifying seconds, microseconds and nanoseconds as business hour results in `ValueError`.

```
In [179]: bh = BusinessHour(start='11:00', end=time(20, 0))

In [180]: bh
Out[180]: <BusinessHour: BH=11:00-20:00>

In [181]: pd.Timestamp('2014-08-01 13:00') + bh
\\Out[181]: Timestamp('2014-08-01 14:00:00')

In [182]: pd.Timestamp('2014-08-01 09:00') + bh
\\Out[182]: Timestamp('2014-08-01 12:00:00')

In [183]: pd.Timestamp('2014-08-01 18:00') + bh
\\Out[183]: Timestamp('2014-08-01 19:00:00')
```

Passing start time later than end represents midnight business hour. In this case, business hour exceeds midnight and overlap to the next day. Valid business hours are distinguished by whether it started from valid `BusinessDay`.

```
In [184]: bh = BusinessHour(start='17:00', end='09:00')

In [185]: bh
Out[185]: <BusinessHour: BH=17:00-09:00>

In [186]: pd.Timestamp('2014-08-01 17:00') + bh
\\Out[186]: Timestamp('2014-08-01 18:00:00')

In [187]: pd.Timestamp('2014-08-01 23:00') + bh
\\Out[187]: Timestamp('2014-08-02 00:00:00')

# Although 2014-08-02 is Saturday,
# it is valid because it starts from 08-01 (Friday).
In [188]: pd.Timestamp('2014-08-02 04:00') + bh
\\Out[188]: Timestamp('2014-08-02 05:00:00')

# Although 2014-08-04 is Monday,
# it is out of business hours because it starts from 08-03 (Sunday).
In [189]: pd.Timestamp('2014-08-04 04:00') + bh
\\Out[189]: Timestamp('2014-08-04 18:00:00')
```

Applying `BusinessHour.rollforward` and `rollback` to out of business hours results in the next business hour start or previous day's end. Different from other offsets, `BusinessHour.rollforward` may output different results from apply by definition.

This is because one day's business hour end is equal to next day's business hour start. For example, under the default business hours (9:00 - 17:00), there is no gap (0 minutes) between 2014-08-01 17:00 and 2014-08-04

09:00.

```
# This adjusts a Timestamp to business hour edge
In [190]: BusinessHour().rollback(pd.Timestamp('2014-08-02 15:00'))
Out[190]: Timestamp('2014-08-01 17:00:00')

In [191]: BusinessHour().rollforward(pd.Timestamp('2014-08-02 15:00'))
\\Out[191]: Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessHour().apply(pd.Timestamp('2014-08-01 17:00')).
# And it is the same as BusinessHour().apply(pd.Timestamp('2014-08-04 09:00'))
In [192]: BusinessHour().apply(pd.Timestamp('2014-08-02 15:00'))
\\Out[192]:
↪Timestamp('2014-08-04 10:00:00')

# BusinessDay results (for reference)
In [193]: BusinessHour().rollforward(pd.Timestamp('2014-08-02'))
\\
↪Timestamp('2014-08-04 09:00:00')

# It is the same as BusinessDay().apply(pd.Timestamp('2014-08-01'))
# The result is the same as rollforward because BusinessDay never overlap.
In [194]: BusinessHour().apply(pd.Timestamp('2014-08-02'))
\\
↪Timestamp('2014-08-04 10:00:00')
```

BusinessHour regards Saturday and Sunday as holidays. To use arbitrary holidays, you can use CustomBusinessHour offset, as explained in the following subsection.

## 19.8.5 Custom Business Hour

New in version 0.18.1.

The CustomBusinessHour is a mixture of BusinessHour and CustomBusinessDay which allows you to specify arbitrary holidays. CustomBusinessHour works as the same as BusinessHour except that it skips specified custom holidays.

```
In [195]: from pandas.tseries.holiday import USFederalHolidayCalendar

In [196]: bhour_us = CustomBusinessHour(calendar=USFederalHolidayCalendar())

# Friday before MLK Day
In [197]: dt = datetime(2014, 1, 17, 15)

In [198]: dt + bhour_us
Out[198]: Timestamp('2014-01-17 16:00:00')

# Tuesday after MLK Day (Monday is skipped because it's a holiday)
In [199]: dt + bhour_us * 2
\\Out[199]: Timestamp('2014-01-21 09:00:00')
```

You can use keyword arguments supported by either BusinessHour and CustomBusinessDay.

```
In [200]: bhour_mon = CustomBusinessHour(start='10:00', weekmask='Tue Wed Thu Fri')

# Monday is skipped because it's a holiday, business hour starts from 10:00
```

(continues on next page)

(continued from previous page)

```
In [201]: dt + bhour_mon * 2
Out [201]: Timestamp('2014-01-21 10:00:00')
```

## 19.8.6 Offset Aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as *offset aliases*.

| Alias    | Description                                      |
|----------|--|
| B        | business day frequency                           |
| C        | custom business day frequency                    |
| D        | calendar day frequency                           |
| W        | weekly frequency                                 |
| M        | month end frequency                              |
| SM       | semi-month end frequency (15th and end of month) |
| BM       | business month end frequency                     |
| CBM      | custom business month end frequency              |
| MS       | month start frequency                            |
| SMS      | semi-month start frequency (1st and 15th)        |
| BMS      | business month start frequency                   |
| CBMS     | custom business month start frequency            |
| Q        | quarter end frequency                            |
| BQ       | business quarter end frequency                   |
| QS       | quarter start frequency                          |
| BQS      | business quarter start frequency                 |
| A, Y     | year end frequency                               |
| BA, BY   | business year end frequency                      |
| AS, YS   | year start frequency                             |
| BAS, BYS | business year start frequency                    |
| BH       | business hour frequency                          |
| H        | hourly frequency                                 |
| T, min   | minutely frequency                               |
| S        | secondly frequency                               |
| L, ms    | milliseconds                                     |
| U, us    | microseconds                                     |
| N        | nanoseconds                                      |

## 19.8.7 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

```
In [202]: pd.date_range(start, periods=5, freq='B')
Out [202]:
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07'],
              dtype='datetime64[ns]', freq='B')

In [203]: pd.date_range(start, periods=5, freq=BDay())
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
DatetimeIndex(['2011-01-03', '2011-01-04', '2011-01-05', '2011-01-06',
               '2011-01-07'],
              dtype='datetime64[ns]', freq='B')
```

You can combine together day and intraday offsets:

```
In [204]: pd.date_range(start, periods=10, freq='2h20min')
```

```
Out[204]:
```

```
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 02:20:00',
               '2011-01-01 04:40:00', '2011-01-01 07:00:00',
               '2011-01-01 09:20:00', '2011-01-01 11:40:00',
               '2011-01-01 14:00:00', '2011-01-01 16:20:00',
               '2011-01-01 18:40:00', '2011-01-01 21:00:00'],
              dtype='datetime64[ns]', freq='140T')
```

```
In [205]: pd.date_range(start, periods=10, freq='1D10U')
```

```
DatetimeIndex([
    '2011-01-01 00:00:00', '2011-01-02 00:00:00.000010',
    '2011-01-03 00:00:00.000020', '2011-01-04 00:00:00.000030',
    '2011-01-05 00:00:00.000040', '2011-01-06 00:00:00.000050',
    '2011-01-07 00:00:00.000060', '2011-01-08 00:00:00.000070',
    '2011-01-09 00:00:00.000080', '2011-01-10 00:00:00.000090'],
              dtype='datetime64[ns]', freq='86400000010U')
```

### 19.8.8 Anchored Offsets

For some frequencies you can specify an anchoring suffix:

| Alias       | Description   |
|-------------|---|
| W-SUN       | weekly frequency (Sundays). Same as 'W'                 |
| W-MON       | weekly frequency (Mondays)                              |
| W-TUE       | weekly frequency (Tuesdays)                             |
| W-WED       | weekly frequency (Wednesdays)                           |
| W-THU       | weekly frequency (Thursdays)                            |
| W-FRI       | weekly frequency (Fridays)                              |
| W-SAT       | weekly frequency (Saturdays)                            |
| (B)Q(S)-DEC | quarterly frequency, year ends in December. Same as 'Q' |
| (B)Q(S)-JAN | quarterly frequency, year ends in January               |
| (B)Q(S)-FEB | quarterly frequency, year ends in February              |
| (B)Q(S)-MAR | quarterly frequency, year ends in March                 |
| (B)Q(S)-APR | quarterly frequency, year ends in April                 |
| (B)Q(S)-MAY | quarterly frequency, year ends in May                   |
| (B)Q(S)-JUN | quarterly frequency, year ends in June                  |

Continued on next page



Table 2 – continued from previous page

| Alias       | Description   |
|-------------|---|
| (B)Q(S)-JUL | quarterly frequency, year ends in July                  |
| (B)Q(S)-AUG | quarterly frequency, year ends in August                |
| (B)Q(S)-SEP | quarterly frequency, year ends in September             |
| (B)Q(S)-OCT | quarterly frequency, year ends in October               |
| (B)Q(S)-NOV | quarterly frequency, year ends in November              |
| (B)A(S)-DEC | annual frequency, anchored end of December. Same as 'A' |
| (B)A(S)-JAN | annual frequency, anchored end of January               |
| (B)A(S)-FEB | annual frequency, anchored end of February              |
| (B)A(S)-MAR | annual frequency, anchored end of March                 |
| (B)A(S)-APR | annual frequency, anchored end of April                 |
| (B)A(S)-MAY | annual frequency, anchored end of May                   |
| (B)A(S)-JUN | annual frequency, anchored end of June                  |
| (B)A(S)-JUL | annual frequency, anchored end of July                  |
| (B)A(S)-AUG | annual frequency, anchored end of August                |
| (B)A(S)-SEP | annual frequency, anchored end of September             |
| (B)A(S)-OCT | annual frequency, anchored end of October               |
| (B)A(S)-NOV | annual frequency, anchored end of November              |

These can be used as arguments to `date_range`, `bdate_range`, constructors for `DatetimeIndex`, as well as various other timeseries-related functions in pandas.

### 19.8.9 Anchored Offset Semantics

For those offsets that are anchored to the start or end of specific frequency (`MonthEnd`, `MonthBegin`, `WeekEnd`, etc), the following rules apply to rolling forward and backwards.

When `n` is not 0, if the given date is not on an anchor point, it snapped to the next(previous) anchor point, and moved `|n| - 1` additional steps forwards or backwards.

```
In [206]: pd.Timestamp('2014-01-02') + MonthBegin(n=1)
Out[206]: Timestamp('2014-02-01 00:00:00')
```

```
In [207]: pd.Timestamp('2014-01-02') + MonthEnd(n=1)
Out[207]: Timestamp('2014-01-31 00:00:00')
```

(continues on next page)

(continued from previous page)

```
In [208]: pd.Timestamp('2014-01-02') - MonthBegin(n=1)
Out[208]:
↳Timestamp('2014-01-01 00:00:00')
```

```
In [209]: pd.Timestamp('2014-01-02') - MonthEnd(n=1)
↳Timestamp('2013-12-31 00:00:00')
```

```
In [210]: pd.Timestamp('2014-01-02') + MonthBegin(n=4)
↳Timestamp('2014-05-01 00:00:00')
```

```
In [211]: pd.Timestamp('2014-01-02') - MonthBegin(n=4)
↳Timestamp('2013-10-01 00:00:00')
```

If the given date *is* on an anchor point, it is moved  $|n|$  points forwards or backwards.

```
In [212]: pd.Timestamp('2014-01-01') + MonthBegin(n=1)
Out[212]: Timestamp('2014-02-01 00:00:00')
```

```
In [213]: pd.Timestamp('2014-01-31') + MonthEnd(n=1)
Out[213]: Timestamp('2014-02-28 00:00:00')
```

```
In [214]: pd.Timestamp('2014-01-01') - MonthBegin(n=1)
Out[214]: Timestamp('2013-12-01 00:00:00')
```

```
In [215]: pd.Timestamp('2014-01-31') - MonthEnd(n=1)
Out[215]: Timestamp('2013-12-31 00:00:00')
```

```
In [216]: pd.Timestamp('2014-01-01') + MonthBegin(n=4)
Out[216]: Timestamp('2014-05-01 00:00:00')
```

```
In [217]: pd.Timestamp('2014-01-31') - MonthBegin(n=4)
Out[217]: Timestamp('2013-10-01 00:00:00')
```

For the case when  $n=0$ , the date is not moved if on an anchor point, otherwise it is rolled forward to the next anchor point.

```
In [218]: pd.Timestamp('2014-01-02') + MonthBegin(n=0)
Out[218]: Timestamp('2014-02-01 00:00:00')

In [219]: pd.Timestamp('2014-01-02') + MonthEnd(n=0)
Out[219]: Timestamp('2014-01-31 00:00:00')

In [220]: pd.Timestamp('2014-01-01') + MonthBegin(n=0)
Out[220]: Timestamp('2014-01-01 00:00:00')

In [221]: pd.Timestamp('2014-01-31') + MonthEnd(n=0)
Out[221]: Timestamp('2014-01-31 00:00:00')
```

## 19.8.10 Holidays / Holiday Calendars

Holidays and calendars provide a simple way to define holiday rules to be used with `CustomBusinessDay` or in other analysis that requires a predefined set of holidays. The `AbstractHolidayCalendar` class provides all the necessary methods to return a list of holidays and only rules need to be defined in a specific holiday calendar class. Furthermore, the `start_date` and `end_date` class attributes determine over what date range holidays are generated. These should be overwritten on the `AbstractHolidayCalendar` class to have the range apply to all calendar subclasses. `USFederalHolidayCalendar` is the only calendar that exists and primarily serves as an example for developing other calendars.

For holidays that occur on fixed dates (e.g., US Memorial Day or July 4th) an observance rule determines when that holiday is observed if it falls on a weekend or some other non-observed day. Defined observance rules are:

| Rule                                | Description  |
|-------------------------------------|--|
| <code>nearest_workday</code>        | move Saturday to Friday and Sunday to Monday         |
| <code>sunday_to_monday</code>       | move Sunday to following Monday                      |
| <code>next_monday_or_tuesday</code> | move Saturday to Monday and Sunday/Monday to Tuesday |
| <code>previous_friday</code>        | move Saturday and Sunday to previous Friday          |
| <code>next_monday</code>            | move Saturday and Sunday to following Monday         |

An example of how holidays and holiday calendars are defined:

```
In [222]: from pandas.tseries.holiday import Holiday, USMemorialDay, \
.....:      AbstractHolidayCalendar, nearest_workday, MO
.....:

In [223]: class ExampleCalendar(AbstractHolidayCalendar):
.....:     rules = [
.....:         USMemorialDay,
.....:         Holiday('July 4th', month=7, day=4, observance=nearest_workday),
.....:         Holiday('Columbus Day', month=10, day=1,
.....:             offset=DateOffset(weekday=MO(2))), #same as 2*Week(weekday=2)
.....:     ]
.....:

In [224]: cal = ExampleCalendar()

In [225]: cal.holidays(datetime(2012, 1, 1), datetime(2012, 12, 31))
Out[225]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype=
->'datetime64[ns]', freq=None)
```

Using this calendar, creating an index or doing offset arithmetic skips weekends and holidays (i.e., Memorial Day/July 4th). For example, the below defines a custom business day offset using the `ExampleCalendar`. Like any other offset, it can be used to create a `DatetimeIndex` or added to `datetime` or `Timestamp` objects.

```
In [226]: from pandas.tseries.offsets import CDay

In [227]: pd.DatetimeIndex(start='7/1/2012', end='7/10/2012',
.....:     freq=CDay(calendar=cal)).to_pydatetime()
.....:
Out[227]:
array([datetime.datetime(2012, 7, 2, 0, 0),
       datetime.datetime(2012, 7, 3, 0, 0),
       datetime.datetime(2012, 7, 5, 0, 0),
       datetime.datetime(2012, 7, 6, 0, 0),
       datetime.datetime(2012, 7, 9, 0, 0),
```

(continues on next page)

(continued from previous page)

[illegible]

Ranges are defined by the `start_date` and `end_date` class attributes of `AbstractHolidayCalendar`. The defaults are shown below.

```
In [233]: AbstractHolidayCalendar.start_date
Out[233]: Timestamp('1970-01-01 00:00:00')
```

```
In [234]: AbstractHolidayCalendar.end_date
Out[234]: Timestamp('2030-12-31 00:00:00')
```

These dates can be overwritten by setting the attributes as datetime/Timestamp/string.

```
In [235]: AbstractHolidayCalendar.start_date = datetime(2012, 1, 1)

In [236]: AbstractHolidayCalendar.end_date = datetime(2012, 12, 31)

In [237]: cal.holidays()
Out[237]: DatetimeIndex(['2012-05-28', '2012-07-04', '2012-10-08'], dtype=
↳ 'datetime64[ns]', freq=None)
```

Every calendar class is accessible by name using the `get_calendar` function which returns a holiday class instance. Any imported calendar class will automatically be available by this function. Also, `HolidayCalendarFactory` provides an easy interface to create calendars that are combinations of calendars or calendars with additional rules.

```
In [238]: from pandas.tseries.holiday import get_calendar, HolidayCalendarFactory, \
.....:      USLaborDay
.....:

In [239]: cal = get_calendar('ExampleCalendar')

In [240]: cal.rules
Out[240]:
[Holiday: MemorialDay (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x1c36e47158>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]

In [241]: new_cal = HolidayCalendarFactory('NewExampleCalendar', cal, USLaborDay)
```

(continues on next page)

(continued from previous page)

```
In [242]: new_cal.rules
Out [242]:
[Holiday: Labor Day (month=9, day=1, offset=<DateOffset: weekday=MO(+1)>),
 Holiday: MemorialDay (month=5, day=31, offset=<DateOffset: weekday=MO(-1)>),
 Holiday: July 4th (month=7, day=4, observance=<function nearest_workday at 0x1c36e47158>),
 Holiday: Columbus Day (month=10, day=1, offset=<DateOffset: weekday=MO(+2)>)]
```

## 19.9 Time Series-Related Instance Methods

### 19.9.1 Shifting / Lagging

One may want to *shift* or *lag* the values in a time series back and forward in time. The method for this is `shift()`, which is available on all of the pandas objects.

```
In [243]: ts = ts[:5]

In [244]: ts.shift(1)
Out [244]:
2011-01-31      NaN
2011-02-28    -1.281247
2011-03-31    -0.727707
2011-04-29    -0.121306
2011-05-31    -0.097883
Freq: BM, dtype: float64
```

The `shift` method accepts an `freq` argument which can accept a `DateOffset` class or other `timedelta`-like object or also an *offset alias*:

```
In [245]: ts.shift(5, freq=offsets.BDay())
Out [245]:
2011-02-07    -1.281247
2011-03-07    -0.727707
2011-04-07    -0.121306
2011-05-06    -0.097883
2011-06-07     0.695775
dtype: float64

In [246]: ts.shift(5, freq='BM')
Out [246]:
2011-06-30    -1.281247
2011-07-29    -0.727707
2011-08-31    -0.121306
2011-09-30    -0.097883
2011-10-31     0.695775
Freq: BM, dtype: float64
```

Rather than changing the alignment of the data and the index, `DataFrame` and `Series` objects also have a `tshift()` convenience method that changes all the dates in the index by a specified number of offsets:

```
In [247]: ts.tshift(5, freq='D')
Out [247]:
```

(continues on next page)

(continued from previous page)

```
2011-02-05    -1.281247
2011-03-05    -0.727707
2011-04-05    -0.121306
2011-05-04    -0.097883
2011-06-05     0.695775
dtype: float64
```

Note that with `tshift`, the leading entry is no longer NaN because the data is not being realigned.

## 19.9.2 Frequency Conversion

The primary function for changing frequencies is the `asfreq()` method. For a `DatetimeIndex`, this is basically just a thin, but convenient wrapper around `reindex()` which generates a `date_range` and calls `reindex`.

```
In [248]: dr = pd.date_range('1/1/2010', periods=3, freq=3 * offsets.BDay())
```

```
In [249]: ts = pd.Series(randn(3), index=dr)
```

```
In [250]: ts
```

Out [250] :

```
2010-01-01    0.155932
2010-01-06    1.486218
2010-01-11   -2.148675
Freq: 3B, dtype: float64
```

```
In [251]: ts.asfreq(BDay())
```

```

↔
2010-01-01    0.155932
2010-01-04         NaN
2010-01-05         NaN
2010-01-06    1.486218
2010-01-07         NaN
2010-01-08         NaN
2010-01-11   -2.148675
Freq: B, dtype: float64

```

`asfreq` provides a further convenience so you can specify an interpolation method for any gaps that may appear after the frequency conversion.

```
In [252]: ts.asfreq(BDay(), method='pad')
```

Out [252] :

```
2010-01-01    0.155932
2010-01-04    0.155932
2010-01-05    0.155932
2010-01-06    1.486218
2010-01-07    1.486218
2010-01-08    1.486218
2010-01-11   -2.148675
Freq: B, dtype: float64
```

### 19.9.3 Filling Forward / Backward

Related to `asfreq` and `reindex` is `fillna()`, which is documented in the *missing data section*.

## 19.9.4 Converting to Python Datetimes

`DatetimeIndex` can be converted to an array of Python native `datetime.datetime` objects using the `to_pydatetime` method.

## 19.10 Resampling

**Warning:** The interface to `.resample` has changed in 0.18.0 to be more groupby-like and hence more flexible. See the [whatsnew docs](#) for a comparison with prior versions.

Pandas has a simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (e.g., converting secondly data into 5-minutely data). This is extremely common in, but not limited to, financial applications.

`resample()` is a time-based groupby, followed by a reduction method on each of its groups. See some [cookbook examples](#) for some advanced strategies.

Starting in version 0.18.1, the `resample()` function can be used directly from `DataFrameGroupBy` objects, see the [groupby docs](#).

**Note:** `.resample()` is similar to using a `rolling()` operation with a time-based offset, see a discussion [here](#).

### 19.10.1 Basics

```
In [253]: rng = pd.date_range('1/1/2012', periods=100, freq='S')
In [254]: ts = pd.Series(np.random.randint(0, 500, len(rng)), index=rng)
In [255]: ts.resample('5Min').sum()
Out[255]:
2012-01-01    25653
Freq: 5T, dtype: int64
```

The `resample` function is very flexible and allows you to specify many different parameters to control the frequency conversion and resampling operation.

Any function available via [dispatching](#) is available as a method of the returned object, including `sum`, `mean`, `std`, `sem`, `max`, `min`, `median`, `first`, `last`, `ohlc`:

```
In [256]: ts.resample('5Min').mean()
Out[256]:
2012-01-01    256.53
Freq: 5T, dtype: float64

In [257]: ts.resample('5Min').ohlc()
Out[257]:
\\Out[257]:
      open  high  low  close
2012-01-01   296   496    6   449

In [258]: ts.resample('5Min').max()
```

(continues on next page)

(continued from previous page)

```

2012-01-01      496
Freq: 5T, dtype: int64

```

For downsampling, `closed` can be set to 'left' or 'right' to specify which end of the interval is closed:

```
In [259]: ts.resample('5Min', closed='right').mean()
Out[259]:
2011-12-31 23:55:00    296.000000
2012-01-01 00:00:00    256.131313
Freq: 5T, dtype: float64

In [260]: ts.resample('5Min', closed='left').mean()
////////////////////////////////////
↪
2012-01-01    256.53
Freq: 5T, dtype: float64
```

Parameters like `label` and `loffset` are used to manipulate the resulting labels. `label` specifies whether the result is labeled with the beginning or the end of the interval. `loffset` performs a time adjustment on the output labels.

```
In [261]: ts.resample('5Min').mean() # by default label='left'
Out[261]:
2012-01-01    256.53
Freq: 5T, dtype: float64

In [262]: ts.resample('5Min', label='left').mean()
Out[262]:
2012-01-01    256.53
Freq: 5T, dtype: float64

In [263]: ts.resample('5Min', label='left', loffset='1s').mean()
Out[263]:
2012-01-01 00:00:01    256.53
dtype: float64
```

**Note:** The default values for `label` and `closed` is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

```
In [264]: rng2 = pd.date_range('1/1/2012', end='3/31/2012', freq='D')

In [265]: ts2 = pd.Series(range(len(rng2)), index=rng2)

# default: label='right', closed='right'
In [266]: ts2.resample('M').max()
Out[266]:
2012-01-31    30
2012-02-29    59
2012-03-31    90
Freq: M, dtype: int64

# default: label='left', closed='left'
In [267]: ts2.resample('SM').max()
```

(continues on next page)



(continued from previous page)

```

\\Out [267]:
↪
2011-12-31      13
2012-01-15      29
2012-01-31      44
2012-02-15      58
2012-02-29      73
2012-03-15      89
2012-03-31      90
Freq: SM-15, dtype: int64

In [268]: ts2.resample('SM', label='right', closed='right').max()
\\
↪
2012-01-15      14.0
2012-01-31      30.0
2012-02-15      45.0
2012-02-29      59.0
2012-03-15      74.0
2012-03-31      90.0
2012-04-15       NaN
Freq: SM-15, dtype: float64

```

The `axis` parameter can be set to 0 or 1 and allows you to resample the specified axis for a `DataFrame`.

`kind` can be set to `'timestamp'` or `'period'` to convert the resulting index to/from timestamp and time span representations. By default `resample` retains the input representation.

`convention` can be set to ‘start’ or ‘end’ when resampling period data (detail below). It specifies how low frequency periods are converted to higher frequency periods.

## 19.10.2 Upsampling

For upsampling, you can specify a way to upsample and the `limit` parameter to interpolate over the gaps that are created:

```
# from secondly to every 250 milliseconds
```

```
In [269]: ts[:2].resample('250L').asfreq()
```

Out [269] :

```
2012-01-01 00:00:00.000    296.0
2012-01-01 00:00:00.250      NaN
2012-01-01 00:00:00.500      NaN
2012-01-01 00:00:00.750      NaN
2012-01-01 00:00:01.000    199.0
Freq: 250L, dtype: float64
```

```
In [270]: ts[:2].resample('250L').ffill()
```

```

2012-01-01 00:00:00.000    296
2012-01-01 00:00:00.250    296
2012-01-01 00:00:00.500    296
2012-01-01 00:00:00.750    296
2012-01-01 00:00:01.000    199
Freq: 250L, dtype: int64

```

(continues on next page)

(continued from previous page)

```
In [271]: ts[:2].resample('250L').ffill(limit=2)
```

```

////////////////////////////////////
↪
2012-01-01 00:00:00.000    296.0
2012-01-01 00:00:00.250    296.0
2012-01-01 00:00:00.500    296.0
2012-01-01 00:00:00.750     NaN
2012-01-01 00:00:01.000    199.0
Freq: 250L, dtype: float64

```

### 19.10.3 Sparse Resampling

Sparse timeseries are the ones where you have a lot fewer points relative to the amount of time you are looking to resample. Naively upsampling a sparse series can potentially generate lots of intermediate values. When you don't want to use a method to fill these values, e.g. `fill_method` is `None`, then intermediate values will be filled with `NaN`.

Since `resample` is a time-based groupby, the following is a method to efficiently resample only the groups that are not all `NaN`.

```
In [272]: rng = pd.date_range('2014-1-1', periods=100, freq='D') + pd.Timedelta('1s')
```

```
In [273]: ts = pd.Series(range(100), index=rng)
```

If we want to resample to the full range of the series:

```
In [274]: ts.resample('3T').sum()
```

```
Out[274]:
```

```

2014-01-01 00:00:00    0
2014-01-01 00:03:00    0
2014-01-01 00:06:00    0
2014-01-01 00:09:00    0
2014-01-01 00:12:00    0
2014-01-01 00:15:00    0
2014-01-01 00:18:00    0
..
2014-04-09 23:42:00    0
2014-04-09 23:45:00    0
2014-04-09 23:48:00    0
2014-04-09 23:51:00    0
2014-04-09 23:54:00    0
2014-04-09 23:57:00    0
2014-04-10 00:00:00    99
Freq: 3T, Length: 47521, dtype: int64

```

We can instead only resample those groups where we have points as follows:

```
In [275]: from functools import partial
```

```
In [276]: from pandas.tseries.frequencies import to_offset
```

```

In [277]: def round(t, freq):
.....:     freq = to_offset(freq)
.....:     return pd.Timestamp((t.value // freq.delta.value) * freq.delta.value)

```

(continues on next page)

(continued from previous page)

```

.....:
In [278]: ts.groupby(partial(round, freq='3T')).sum()
Out[278]:
2014-01-01      0
2014-01-02      1
2014-01-03      2
2014-01-04      3
2014-01-05      4
2014-01-06      5
2014-01-07      6
..
2014-04-04     93
2014-04-05     94
2014-04-06     95
2014-04-07     96
2014-04-08     97
2014-04-09     98
2014-04-10     99
Length: 100, dtype: int64

```

### 19.10.4 Aggregation

Similar to the *aggregating API*, *groupby API*, and the *window functions API*, a Resampler can be selectively resampled.

Resampling a DataFrame, the default will be to act on all columns with the same function.

```

In [279]: df = pd.DataFrame(np.random.randn(1000, 3),
.....:                      index=pd.date_range('1/1/2012', freq='S', periods=1000),
.....:                      columns=['A', 'B', 'C'])
.....:

In [280]: r = df.resample('3T')

In [281]: r.mean()
Out[281]:
              A              B              C
2012-01-01 00:00:00 -0.038580 -0.085117 -0.024750
2012-01-01 00:03:00  0.052387 -0.061477  0.029548
2012-01-01 00:06:00  0.121377 -0.010630 -0.043691
2012-01-01 00:09:00 -0.106814 -0.053819  0.097222
2012-01-01 00:12:00  0.032560  0.080543  0.167380
2012-01-01 00:15:00  0.060486 -0.057602 -0.106213

```

We can select a specific column or columns using standard getitem.

```

In [282]: r['A'].mean()
Out[282]:
2012-01-01 00:00:00 -0.038580
2012-01-01 00:03:00  0.052387
2012-01-01 00:06:00  0.121377
2012-01-01 00:09:00 -0.106814
2012-01-01 00:12:00  0.032560
2012-01-01 00:15:00  0.060486

```

(continues on next page)

(continued from previous page)

Freq: 3T, Name: A, dtype: float64

**In [283]:** r[['A', 'B']].mean()

```

////////////////////////////////////
↪

```

|                     | A         | B         |
|---------------------|-----------|-----------|
| 2012-01-01 00:00:00 | -0.038580 | -0.085117 |
| 2012-01-01 00:03:00 | 0.052387  | -0.061477 |
| 2012-01-01 00:06:00 | 0.121377  | -0.010630 |
| 2012-01-01 00:09:00 | -0.106814 | -0.053819 |
| 2012-01-01 00:12:00 | 0.032560  | 0.080543  |
| 2012-01-01 00:15:00 | 0.060486  | -0.057602 |

You can pass a list or dict of functions to do aggregation with, outputting a DataFrame:

**In [284]:** r['A'].agg([np.sum, np.mean, np.std])**Out[284]:**

|                     | sum        | mean      | std      |
|---------------------|------------|-----------|----------|
| 2012-01-01 00:00:00 | -6.944481  | -0.038580 | 0.985150 |
| 2012-01-01 00:03:00 | 9.429707   | 0.052387  | 1.078022 |
| 2012-01-01 00:06:00 | 21.847876  | 0.121377  | 0.996365 |
| 2012-01-01 00:09:00 | -19.226593 | -0.106814 | 0.914070 |
| 2012-01-01 00:12:00 | 5.860874   | 0.032560  | 1.100055 |
| 2012-01-01 00:15:00 | 6.048588   | 0.060486  | 1.001532 |

On a resampled DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index:

**In [285]:** r.agg([np.sum, np.mean])**Out[285]:**

|                     | A          |           | B          |           | C          |           |
|---------------------|------------|-----------|------------|-----------|------------|-----------|
|                     | sum        | mean      | sum        | mean      | sum        | mean      |
| 2012-01-01 00:00:00 | -6.944481  | -0.038580 | -15.320993 | -0.085117 | -4.454941  | -0.024750 |
| 2012-01-01 00:03:00 | 9.429707   | 0.052387  | -11.065916 | -0.061477 | 5.318688   | 0.029548  |
| 2012-01-01 00:06:00 | 21.847876  | 0.121377  | -1.913420  | -0.010630 | -7.864429  | -0.043691 |
| 2012-01-01 00:09:00 | -19.226593 | -0.106814 | -9.687468  | -0.053819 | 17.499920  | 0.097222  |
| 2012-01-01 00:12:00 | 5.860874   | 0.032560  | 14.497725  | 0.080543  | 30.128432  | 0.167380  |
| 2012-01-01 00:15:00 | 6.048588   | 0.060486  | -5.760208  | -0.057602 | -10.621260 | -0.106213 |

By passing a dict to aggregate you can apply a different aggregation to the columns of a DataFrame:

```

In [286]: r.agg({'A' : np.sum,
.....:          'B' : lambda x: np.std(x, ddof=1)})
.....:

```

**Out[286]:**

|                     | A          | B        |
|---------------------|------------|----------|
| 2012-01-01 00:00:00 | -6.944481  | 1.087752 |
| 2012-01-01 00:03:00 | 9.429707   | 1.014552 |
| 2012-01-01 00:06:00 | 21.847876  | 0.954588 |
| 2012-01-01 00:09:00 | -19.226593 | 1.027990 |
| 2012-01-01 00:12:00 | 5.860874   | 1.021503 |
| 2012-01-01 00:15:00 | 6.048588   | 1.004984 |

The function names can also be strings. In order for a string to be valid it must be implemented on the resampled object:

```
In [287]: r.agg({'A' : 'sum', 'B' : 'std'})
Out[287]:
```

|                     | A          | B        |
|---------------------|------------|----------|
| 2012-01-01 00:00:00 | -6.944481  | 1.087752 |
| 2012-01-01 00:03:00 | 9.429707   | 1.014552 |
| 2012-01-01 00:06:00 | 21.847876  | 0.954588 |
| 2012-01-01 00:09:00 | -19.226593 | 1.027990 |
| 2012-01-01 00:12:00 | 5.860874   | 1.021503 |
| 2012-01-01 00:15:00 | 6.048588   | 1.004984 |

Furthermore, you can also specify multiple aggregation functions for each column separately.

```
In [288]: r.agg({'A' : ['sum', 'std'], 'B' : ['mean', 'std'] })
Out[288]:
```

|                     | A          |          | B         |          |
|---------------------|------------|----------|-----------|----------|
|                     | sum        | std      | mean      | std      |
| 2012-01-01 00:00:00 | -6.944481  | 0.985150 | -0.085117 | 1.087752 |
| 2012-01-01 00:03:00 | 9.429707   | 1.078022 | -0.061477 | 1.014552 |
| 2012-01-01 00:06:00 | 21.847876  | 0.996365 | -0.010630 | 0.954588 |
| 2012-01-01 00:09:00 | -19.226593 | 0.914070 | -0.053819 | 1.027990 |
| 2012-01-01 00:12:00 | 5.860874   | 1.100055 | 0.080543  | 1.021503 |
| 2012-01-01 00:15:00 | 6.048588   | 1.001532 | -0.057602 | 1.004984 |

If a DataFrame does not have a datetimelike index, but instead you want to resample based on datetimelike column in the frame, it can be passed to the `on` keyword.

```
In [289]: df = pd.DataFrame({'date': pd.date_range('2015-01-01', freq='W', periods=5),
.....:                      'a': np.arange(5)},
.....:                      index=pd.MultiIndex.from_arrays([
.....:                          [1,2,3,4,5],
.....:                          pd.date_range('2015-01-01', freq='W',
.....:                                          periods=5)],
.....:                      names=['v', 'd']))
```

```
In [290]: df
Out[290]:
```

|     | date                  | a |
|-----|-----------------------|---|
| v d |                       |   |
| 1   | 2015-01-04 2015-01-04 | 0 |
| 2   | 2015-01-11 2015-01-11 | 1 |
| 3   | 2015-01-18 2015-01-18 | 2 |
| 4   | 2015-01-25 2015-01-25 | 3 |
| 5   | 2015-02-01 2015-02-01 | 4 |

```
In [291]: df.resample('M', on='date').sum()
```

```

////////////////////////////////////
↪
a
date
2015-01-31  6
2015-02-28  4
```

Similarly, if you instead want to resample by a datetimelike level of `MultiIndex`, its name or location can be passed to the `level` keyword.

```
In [292]: df.resample('M', level='d').sum()
```

(continues on next page)

(continued from previous page)

```
Out [292]:
          a
d
2015-01-31  6
2015-02-28  4
```

## 19.11 Time Span Representation

Regular intervals of time are represented by `Period` objects in pandas while sequences of `Period` objects are collected in a `PeriodIndex`, which can be created with the convenience function `period_range`.

### 19.11.1 Period

A `Period` represents a span of time (e.g., a day, a month, a quarter, etc). You can specify the span via `freq` keyword using a frequency alias like below. Because `freq` represents a span of `Period`, it cannot be negative like “-3D”.

```
In [293]: pd.Period('2012', freq='A-DEC')
Out [293]: Period('2012', 'A-DEC')

In [294]: pd.Period('2012-1-1', freq='D')
Out [294]: Period('2012-01-01', 'D')

In [295]: pd.Period('2012-1-1 19:00', freq='H')
Out [295]: Period('2012-01-01 19:00', 'H')

In [296]: pd.Period('2012-1-1 19:00', freq='5H')
Out [296]: Period('2012-01-01 19:00', '5H')
```

Adding and subtracting integers from periods shifts the period by its own frequency. Arithmetic is not allowed between `Period` with different `freq` (span).

```
In [297]: p = pd.Period('2012', freq='A-DEC')

In [298]: p + 1
Out [298]: Period('2013', 'A-DEC')

In [299]: p - 3
Out [299]: Period('2009', 'A-DEC')

In [300]: p = pd.Period('2012-01', freq='2M')

In [301]: p + 2
Out [301]: Period('2012-05', '2M')

In [302]: p - 1
Out [302]: Period('2011-11', '2M')

In [303]: p == pd.Period('2012-01', freq='3M')
Out [303]:
IncompatibleFrequency                                Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-303-4b67dc0b596c> in <module>()
----> 1 p == pd.Period('2012-01', freq='3M')

~/sandbox/pandas-release/pandas-docs/pandas/_libs/tslibs/period.pyx in pandas._libs.
↳tslibs.period._Period.__richcmp__()

IncompatibleFrequency: Input has different freq=3M from Period(freq=2M)
```

If Period freq is daily or higher (D, H, T, S, L, U, N), offsets and timedelta-like can be added if the result can have the same freq. Otherwise, ValueError will be raised.

[illegible]

```
In [1]: p + Minute(5)
Traceback
...
ValueError: Input has different freq from Period(freq=H)
```

If `Period` has other freqs, only the same offsets can be added. Otherwise, `ValueError` will be raised.

```
In [308]: p = pd.Period('2014-07', freq='M')

In [309]: p + MonthEnd(3)
Out[309]: Period('2014-10', 'M')
```

```
In [1]: p + MonthBegin(3)
Traceback
...
ValueError: Input has different freq from Period(freq=M)
```

Taking the difference of `Period` instances with the same frequency will return the number of frequency units between them:

```
In [310]: pd.Period('2012', freq='A-DEC') - pd.Period('2002', freq='A-DEC')
Out[310]: 10
```

### 19.11.2 PeriodIndex and period\_range

Regular sequences of `Period` objects can be collected in a `PeriodIndex`, which can be constructed using the `period_range` convenience function:

```
In [311]: prng = pd.period_range('1/1/2011', '1/1/2012', freq='M')
In [312]: prng
```

(continues on next page)

(continued from previous page)

```
Out [312]:
PeriodIndex(['2011-01', '2011-02', '2011-03', '2011-04', '2011-05', '2011-06',
            '2011-07', '2011-08', '2011-09', '2011-10', '2011-11', '2011-12',
            '2012-01'],
            dtype='period[M]', freq='M')
```

The PeriodIndex constructor can also be used directly:

```
In [313]: pd.PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out [313]: PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]', freq='M')
```

Passing multiplied frequency outputs a sequence of Period which has multiplied span.

```
In [314]: pd.PeriodIndex(start='2014-01', freq='3M', periods=4)
Out [314]: PeriodIndex(['2014-01', '2014-04', '2014-07', '2014-10'], dtype='period[3M]',
                        freq='3M')
```

If start or end are Period objects, they will be used as anchor endpoints for a PeriodIndex with frequency matching that of the PeriodIndex constructor.

```
In [315]: pd.PeriodIndex(start=pd.Period('2017Q1', freq='Q'),
                        .....: end=pd.Period('2017Q2', freq='Q'), freq='M')
Out [315]: PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'], dtype='period[M]',
                        freq='M')
```

Just like DatetimeIndex, a PeriodIndex can also be used to index pandas objects:

```
In [316]: ps = pd.Series(np.random.randn(len(prng)), prng)

In [317]: ps
Out [317]:
2011-01    0.258318
2011-02   -2.503700
2011-03   -0.303053
2011-04    0.270509
2011-05    1.004841
2011-06   -0.129044
2011-07   -1.406335
2011-08   -1.310412
2011-09    0.769439
2011-10   -0.542325
2011-11    2.010541
2011-12    1.001558
2012-01   -0.087453
Freq: M, dtype: float64
```

PeriodIndex supports addition and subtraction with the same rule as Period.

```
In [318]: idx = pd.period_range('2014-07-01 09:00', periods=5, freq='H')

In [319]: idx
Out [319]:
PeriodIndex(['2014-07-01 09:00', '2014-07-01 10:00', '2014-07-01 11:00',
            '2014-07-01 12:00', '2014-07-01 13:00'],
            dtype='period[H]', freq='H')
```

(continues on next page)



(continued from previous page)

PeriodIndex has its own dtype named `period`, refer to *Period Dtypes*.

### 19.11.3 Period Dtypes

New in version 0.19.0.

`PeriodIndex` has a custom `period` dtype. This is a pandas extension dtype similar to the *timezone aware dtype* (`datetime64[ns, tz]`).

The period dtype holds the freq attribute and is represented with period[freq] like period[D] or period[M], using *frequency strings*.

```
In [324]: pi = pd.period_range('2016-01-01', periods=3, freq='M')

In [325]: pi
Out[325]: PeriodIndex(['2016-01', '2016-02', '2016-03'], dtype='period[M]', freq='M')

In [326]: pi.dtype
\\Out[326]: period[M]
```

The period dtype can be used in `.astype(...)`. It allows one to change the `freq` of a `PeriodIndex` like `.asfreq()` and convert a `DatetimeIndex` to `PeriodIndex` like `to_period()`:

```
# change monthly freq to daily freq
In [327]: pi.astype('period[D]')
Out[327]: PeriodIndex(['2016-01-31', '2016-02-29', '2016-03-31'], dtype='period[D]',
→freq='D')

# convert to DatetimeIndex
In [328]: pi.astype('datetime64[ns]')
Out[328]: DatetimeIndex(['2016-01-01', '2016-02-01', '2016-03-01'], dtype='datetime64[ns]',
→freq='MS')

# convert to PeriodIndex
In [329]: dti = pd.date_range('2011-01-01', freq='M', periods=3)
```

(continues on next page)

(continued from previous page)

```
In [330]: dti
Out[330]: DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31'], dtype=
↳ 'datetime64[ns]', freq='M')

In [331]: dti.astype('period[M]')
////////////////////////////////////
↳ PeriodIndex(['2011-01', '2011-02', '2011-03'], dtype='period[M]', freq='M')
```

#### 19.11.4 PeriodIndex Partial String Indexing

You can pass in dates and strings to `Series` and `DataFrame` with `PeriodIndex`, in the same manner as `DatetimeIndex`. For details, refer to [DatetimeIndex Partial String Indexing](#).

```
In [332]: ps['2011-01']
Out[332]: 0.25831819727391592

In [333]: ps[datetime(2011, 12, 25):]
Out[333]:
2011-12      1.001558
2012-01     -0.087453
Freq: M, dtype: float64

In [334]: ps['10/31/2011':'12/31/2011']
Out[334]:
↔
2011-10     -0.542325
2011-11      2.010541
2011-12      1.001558
Freq: M, dtype: float64
```

Passing a string representing a lower frequency than `PeriodIndex` returns partial sliced data.

```
In [335]: ps['2011']
Out[335]:
2011-01    0.258318
2011-02   -2.503700
2011-03   -0.303053
2011-04    0.270509
2011-05    1.004841
2011-06   -0.129044
2011-07   -1.406335
2011-08   -1.310412
2011-09    0.769439
2011-10   -0.542325
2011-11    2.010541
2011-12    1.001558
Freq: M, dtype: float64

In [336]: dfp = pd.DataFrame(np.random.randn(600,1),
.....:                        columns=['A'],
.....:                        index=pd.period_range('2013-01-01 9:00', periods=600,
↳ freq='T'))
.....:
```

(continues on next page)

(continued from previous page)

**In [337]:** dfp**Out [337]:**

```

          A
2013-01-01 09:00  0.005210
2013-01-01 09:01 -0.014385
2013-01-01 09:02 -0.212404
2013-01-01 09:03 -1.227760
2013-01-01 09:04 -0.809722
2013-01-01 09:05 -1.719723
2013-01-01 09:06 -0.808486
...
2013-01-01 18:53 -0.783098
2013-01-01 18:54  0.755005
2013-01-01 18:55 -1.116732
2013-01-01 18:56 -0.940692
2013-01-01 18:57  0.228536
2013-01-01 18:58  0.109472
2013-01-01 18:59  0.235414

```

[600 rows x 1 columns]

**In [338]:** dfp['2013-01-01 10H']

```

////////////////////////////////////

```

↪

```

          A
2013-01-01 10:00 -0.148998
2013-01-01 10:01  2.154810
2013-01-01 10:02 -1.605646
2013-01-01 10:03  0.021024
2013-01-01 10:04 -0.623737
2013-01-01 10:05  1.451612
2013-01-01 10:06  1.062463
...
2013-01-01 10:53  0.273119
2013-01-01 10:54 -0.994071
2013-01-01 10:55 -1.222179
2013-01-01 10:56 -1.167118
2013-01-01 10:57  0.262822
2013-01-01 10:58 -0.283786
2013-01-01 10:59  1.190726

```

[60 rows x 1 columns]

As with `DatetimeIndex`, the endpoints will be included in the result. The example below slices data starting from 10:00 to 11:59.

**In [339]:** dfp['2013-01-01 10H':'2013-01-01 11H']**Out [339]:**

```

          A
2013-01-01 10:00 -0.148998
2013-01-01 10:01  2.154810
2013-01-01 10:02 -1.605646
2013-01-01 10:03  0.021024
2013-01-01 10:04 -0.623737
2013-01-01 10:05  1.451612
2013-01-01 10:06  1.062463

```

(continues on next page)

(continued from previous page)

```

...
2013-01-01 11:53 -1.477914
2013-01-01 11:54  0.594465
2013-01-01 11:55 -0.903243
2013-01-01 11:56  1.182131
2013-01-01 11:57  0.621345
2013-01-01 11:58 -0.996113
2013-01-01 11:59 -0.191659

[120 rows x 1 columns]

```

### 19.11.5 Frequency Conversion and Resampling with PeriodIndex

The frequency of `Period` and `PeriodIndex` can be converted via the `asfreq` method. Let's start with the fiscal year 2011, ending in December:

```

In [340]: p = pd.Period('2011', freq='A-DEC')

In [341]: p
Out[341]: Period('2011', 'A-DEC')

```

We can convert it to a monthly frequency. Using the `how` parameter, we can specify whether to return the starting or ending month:

```

In [342]: p.asfreq('M', how='start')
Out[342]: Period('2011-01', 'M')

In [343]: p.asfreq('M', how='end')
Out[343]: Period('2011-12', 'M')

```

The shorthands 's' and 'e' are provided for convenience:

```

In [344]: p.asfreq('M', 's')
Out[344]: Period('2011-01', 'M')

In [345]: p.asfreq('M', 'e')
Out[345]: Period('2011-12', 'M')

```

Converting to a “super-period” (e.g., annual frequency is a super-period of quarterly frequency) automatically returns the super-period that includes the input period:

```

In [346]: p = pd.Period('2011-12', freq='M')

In [347]: p.asfreq('A-NOV')
Out[347]: Period('2012', 'A-NOV')

```

Note that since we converted to an annual frequency that ends the year in November, the monthly period of December 2011 is actually in the 2012 A-NOV period.

Period conversions with anchored frequencies are particularly useful for working with various quarterly data common to economics, business, and other fields. Many organizations define quarters relative to the month in which their fiscal year starts and ends. Thus, first quarter of 2011 could start in 2010 or a few months into 2011. Via anchored frequencies, pandas works for all quarterly frequencies Q-JAN through Q-DEC.

Q-DEC define regular calendar quarters:

```
In [351]: p = pd.Period('2011Q4', freq='Q-MAR')

In [352]: p.asfreq('D', 's')
Out[352]: Period('2011-01-01', 'D')

In [353]: p.asfreq('D', 'e')
Out[353]: Period('2011-03-31', 'D')
```

Timestamped data can be converted to PeriodIndex-ed data using `to_period` and vice-versa using `to_timestamp`:

```
In [354]: rng = pd.date_range('1/1/2012', periods=5, freq='M')

In [355]: ts = pd.Series(np.random.randn(len(rng)), index=rng)

In [356]: ts
Out[356]:
2012-01-31    -0.898547
2012-02-29    -1.332247
2012-03-31    -0.741645
2012-04-30     0.094321
2012-05-31    -0.438813
Freq: M, dtype: float64

In [357]: ps = ts.to_period()

In [358]: ps
Out[358]:
2012-01    -0.898547
2012-02    -1.332247
2012-03    -0.741645
2012-04     0.094321
2012-05    -0.438813
Freq: M, dtype: float64

In [359]: ps.to_timestamp()
```

↪

```
2012-01-01    -0.898547
2012-02-01    -1.332247
2012-03-01    -0.741645
2012-04-01     0.094321
```

## 19.12. Converting Between Representations

(continued from previous page)

```
2012-05-01    -0.438813
Freq: MS, dtype: float64
```

Remember that 's' and 'e' can be used to return the timestamps at the start or end of the period:

```
In [360]: ps.to_timestamp('D', how='s')
Out[360]:
2012-01-01    -0.898547
2012-02-01    -1.332247
2012-03-01    -0.741645
2012-04-01     0.094321
2012-05-01    -0.438813
Freq: MS, dtype: float64
```

Converting between period and timestamp enables some convenient arithmetic functions to be used. In the following example, we convert a quarterly frequency with year ending in November to 9am of the end of the month following the quarter end:

```
In [361]: prng = pd.period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [362]: ts = pd.Series(np.random.randn(len(prng)), prng)

In [363]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [364]: ts.head()
Out[364]:
1990-03-01 09:00    -0.564874
1990-06-01 09:00    -1.426510
1990-09-01 09:00     1.295437
1990-12-01 09:00     1.124017
1991-03-01 09:00     0.840428
Freq: H, dtype: float64
```

## 19.13 Representing Out-of-Bounds Spans

If you have data that is outside of the Timestamp bounds, see [Timestamp limitations](#), then you can use a PeriodIndex and/or Series of Periods to do computations.

```
In [365]: span = pd.period_range('1215-01-01', '1381-01-01', freq='D')

In [366]: span
Out[366]:
PeriodIndex(['1215-01-01', '1215-01-02', '1215-01-03', '1215-01-04',
            '1215-01-05', '1215-01-06', '1215-01-07', '1215-01-08',
            '1215-01-09', '1215-01-10',
            ...,
            '1380-12-23', '1380-12-24', '1380-12-25', '1380-12-26',
            '1380-12-27', '1380-12-28', '1380-12-29', '1380-12-30',
            '1380-12-31', '1381-01-01'],
            dtype='period[D]', length=60632, freq='D')
```

To convert from an int64 based YYYYMMDD representation.

[illegible]

These can easily be converted to a `PeriodIndex`:

```
In [372]: span = pd.PeriodIndex(s.apply(conv))

In [373]: span
Out[373]: PeriodIndex(['2012-12-31', '2014-11-30', '9999-12-31'], dtype='period[D]',
    freq='D')
```

## 19.14 Time Zone Handling

Pandas provides rich support for working with timestamps in different time zones using `pytz` and `dateutil` libraries. `dateutil` currently is only supported for fixed offset and `tzfile` zones. The default library is `pytz`. Support for `dateutil` is provided for compatibility with other applications e.g. if you use `dateutil` in other Python packages.

### 19.14.1 Working with Time Zones

By default, pandas objects are time zone unaware:

```
In [374]: rng = pd.date_range('3/6/2012 00:00', periods=15, freq='D')

In [375]: rng.tz is None
Out[375]: True
```

To supply the time zone, you can use the `tz` keyword to `date_range` and other functions. Dateutil time zone strings are distinguished from `pytz` time zones by starting with `dateutil/`.

- In `pytz` you can find a list of common (and less common) time zones using `from pytz import common_timezones, all_timezones`.

- `dateutil` uses the OS timezones so there isn't a fixed list available. For common zones, the names are the same as `pytz`.

```
# pytz
In [376]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
.....:                             tz='Europe/London')
.....:

In [377]: rng_pytz.tz
Out[377]: <DstTzInfo 'Europe/London' LMT-1 day, 23:59:00 STD>

# dateutil
In [378]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
.....:                                 tz='dateutil/Europe/London')
.....:

In [379]: rng_dateutil.tz
Out[379]: tzfile('/usr/share/zoneinfo/Europe/London')

# dateutil - utc special case
In [380]: rng_utc = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
.....:                             tz=dateutil.tz.tzutc())
.....:

In [381]: rng_utc.tz
Out[381]: tzutc()
```

Note that the UTC timezone is a special case in `dateutil` and should be constructed explicitly as an instance of `dateutil.tz.tzutc`. You can also construct other timezones explicitly first, which gives you more control over which time zone is used:

```
# pytz
In [382]: tz_pytz = pytz.timezone('Europe/London')

In [383]: rng_pytz = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
.....:                             tz=tz_pytz)
.....:

In [384]: rng_pytz.tz == tz_pytz
Out[384]: True

# dateutil
In [385]: tz_dateutil = dateutil.tz.gettz('Europe/London')

In [386]: rng_dateutil = pd.date_range('3/6/2012 00:00', periods=10, freq='D',
.....:                                 tz=tz_dateutil)
.....:

In [387]: rng_dateutil.tz == tz_dateutil
Out[387]: True
```

Timestamps, like Python's `datetime.datetime` object can be either time zone naive or time zone aware. Naive time series and `DatetimeIndex` objects can be *localized* using `tz_localize`:

```
In [388]: ts = pd.Series(np.random.randn(len(rng)), rng)

In [389]: ts_utc = ts.tz_localize('UTC')
```

(continues on next page)



(continued from previous page)

```
In [390]: ts_utc
Out[390]:
2012-03-06 00:00:00+00:00    0.037206
2012-03-07 00:00:00+00:00    2.313998
2012-03-08 00:00:00+00:00    1.458296
2012-03-09 00:00:00+00:00   -0.620431
2012-03-10 00:00:00+00:00   -0.000111
2012-03-11 00:00:00+00:00   -0.342783
2012-03-12 00:00:00+00:00   -0.664322
2012-03-13 00:00:00+00:00    0.654814
2012-03-14 00:00:00+00:00    1.550680
2012-03-15 00:00:00+00:00    0.174511
2012-03-16 00:00:00+00:00    1.360491
2012-03-17 00:00:00+00:00    0.799737
2012-03-18 00:00:00+00:00    0.449149
2012-03-19 00:00:00+00:00    0.111346
2012-03-20 00:00:00+00:00   -0.435531
Freq: D, dtype: float64
```

Again, you can explicitly construct the timezone object first. You can use the `tz_convert` method to convert pandas objects to convert tz-aware data to another time zone:

```
In [391]: ts_utc.tz_convert('US/Eastern')
Out[391]:
2012-03-05 19:00:00-05:00    0.037206
2012-03-06 19:00:00-05:00    2.313998
2012-03-07 19:00:00-05:00    1.458296
2012-03-08 19:00:00-05:00   -0.620431
2012-03-09 19:00:00-05:00   -0.000111
2012-03-10 19:00:00-05:00   -0.342783
2012-03-11 20:00:00-04:00   -0.664322
2012-03-12 20:00:00-04:00    0.654814
2012-03-13 20:00:00-04:00    1.550680
2012-03-14 20:00:00-04:00    0.174511
2012-03-15 20:00:00-04:00    1.360491
2012-03-16 20:00:00-04:00    0.799737
2012-03-17 20:00:00-04:00    0.449149
2012-03-18 20:00:00-04:00    0.111346
2012-03-19 20:00:00-04:00   -0.435531
Freq: D, dtype: float64
```

**Warning:** Be wary of conversions between libraries. For some zones `pytz` and `dateutil` have different definitions of the zone. This is more of a problem for unusual timezones than for ‘standard’ zones like US/Eastern.

**Warning:** Be aware that a timezone definition across versions of timezone libraries may not be considered equal. This may cause problems when working with stored data that is localized using one version and operated on with a different version. See [here](#) for how to handle such a situation.

**Warning:** It is incorrect to pass a timezone directly into the `datetime.datetime` constructor (e.g., `datetime.datetime(2011, 1, 1, tz=timezone('US/Eastern'))`). Instead, the `datetime` needs

to be localized using the localize method on the timezone.

Under the hood, all timestamps are stored in UTC. Scalar values from a `DatetimeIndex` with a time zone will have their fields (day, hour, minute) localized to the time zone. However, timestamps with the same UTC value are still considered to be equal even if they are in different time zones:

```
In [392]: rng_eastern = rng_utc.tz_convert('US/Eastern')

In [393]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')

In [394]: rng_eastern[5]
Out[394]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', freq='D')

In [395]: rng_berlin[5]
Out[395]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', freq='D')

In [396]: rng_eastern[5] == rng_berlin[5]
Out[396]: True
```

Like `Series`, `DataFrame`, and `DatetimeIndex`, `Timestamp`'s can be converted to other time zones using `tz_convert`:

```
In [397]: rng_eastern[5]
Out[397]: Timestamp('2012-03-10 19:00:00-0500', tz='US/Eastern', freq='D')

In [398]: rng_berlin[5]
Out[398]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin', freq='D')

In [399]: rng_eastern[5].tz_convert('Europe/Berlin')
Out[399]: Timestamp('2012-03-11 01:00:00+0100', tz='Europe/Berlin')
```

Localization of `Timestamp` functions just like `DatetimeIndex` and `Series`:

```
In [400]: rng[5]
Out[400]: Timestamp('2012-03-11 00:00:00', freq='D')

In [401]: rng[5].tz_localize('Asia/Shanghai')
Out[401]: Timestamp('2012-03-11 00:00:00+0800', tz='Asia/Shanghai')
```

Operations between `Series` in different time zones will yield UTC `Series`, aligning the data on the UTC timestamps:

```
In [402]: eastern = ts_utc.tz_convert('US/Eastern')

In [403]: berlin = ts_utc.tz_convert('Europe/Berlin')

In [404]: result = eastern + berlin

In [405]: result
Out[405]:
2012-03-06 00:00:00+00:00    0.074412
```

(continues on next page)

(continued from previous page)

```

2012-03-07 00:00:00+00:00    4.627997
2012-03-08 00:00:00+00:00    2.916592
2012-03-09 00:00:00+00:00   -1.240863
2012-03-10 00:00:00+00:00   -0.000221
2012-03-11 00:00:00+00:00   -0.685566
2012-03-12 00:00:00+00:00   -1.328643
2012-03-13 00:00:00+00:00    1.309628
2012-03-14 00:00:00+00:00    3.101359
2012-03-15 00:00:00+00:00    0.349022
2012-03-16 00:00:00+00:00    2.720983
2012-03-17 00:00:00+00:00    1.599475
2012-03-18 00:00:00+00:00    0.898297
2012-03-19 00:00:00+00:00    0.222691
2012-03-20 00:00:00+00:00   -0.871062
Freq: D, dtype: float64

```

```
In [406]: result.index
```

```

DatetimeIndex(['2012-03-06', '2012-03-07', '2012-03-08', '2012-03-09',
              '2012-03-10', '2012-03-11', '2012-03-12', '2012-03-13',
              '2012-03-14', '2012-03-15', '2012-03-16', '2012-03-17',
              '2012-03-18', '2012-03-19', '2012-03-20'],
              dtype='datetime64[ns, UTC]', freq='D')

```

To remove timezone from tz-aware DatetimeIndex, use `tz_localize(None)` or `tz_convert(None)`. `tz_localize(None)` will remove timezone holding local time representations. `tz_convert(None)` will remove timezone after converting to UTC time.

```
In [407]: didx = pd.DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz=
      ↪ 'US/Eastern')
```

```
In [408]: didx
```

```

Out[408]:
DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
              '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
              '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
              '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
              '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

```

```
In [409]: didx.tz_localize(None)
```

```

DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
              '2014-08-01 11:00:00', '2014-08-01 12:00:00',
              '2014-08-01 13:00:00', '2014-08-01 14:00:00',
              '2014-08-01 15:00:00', '2014-08-01 16:00:00',
              '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
              dtype='datetime64[ns]', freq='H')

```

```
In [410]: didx.tz_convert(None)
```

```

DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
              '2014-08-01 15:00:00', '2014-08-01 16:00:00'],
              dtype='datetime64[ns]', freq='H')

```

(continues on next page)

(continued from previous page)

```

        '2014-08-01 17:00:00', '2014-08-01 18:00:00',
        '2014-08-01 19:00:00', '2014-08-01 20:00:00',
        '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
        dtype='datetime64[ns]', freq='H')

# tz_convert(None) is identical with tz_convert('UTC').tz_localize(None)
In [411]: didx.tz_convert('UCT').tz_localize(None)
///////////////////////////////////////////////////
↪
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
                '2014-08-01 15:00:00', '2014-08-01 16:00:00',
                '2014-08-01 17:00:00', '2014-08-01 18:00:00',
                '2014-08-01 19:00:00', '2014-08-01 20:00:00',
                '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
                dtype='datetime64[ns]', freq='H')

```

### 19.14.2 Ambiguous Times when Localizing

In some cases, localize cannot determine the DST and non-DST hours when there are duplicates. This often happens when reading files or database records that simply duplicate the hours. Passing `ambiguous='infer'` into `tz_localize` will attempt to determine the right offset. Below the top example will fail as it contains ambiguous times and the bottom will infer the right offset.

```

In [412]: rng_hourly = pd.DatetimeIndex(['11/06/2011 00:00', '11/06/2011 01:00',
.....:                                '11/06/2011 01:00', '11/06/2011 02:00',
.....:                                '11/06/2011 03:00'])
.....:

```

This will fail as there are ambiguous times

```

In [2]: rng_hourly.tz_localize('US/Eastern')
AmbiguousTimeError: Cannot infer dst time from Timestamp('2011-11-06 01:00:00'), try_
↪ using the 'ambiguous' argument

```

Infer the ambiguous times

```

In [413]: rng_hourly_eastern = rng_hourly.tz_localize('US/Eastern', ambiguous='infer')

In [414]: rng_hourly_eastern.tolist()
Out[414]:
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]

```

In addition to ‘infer’, there are several other arguments supported. Passing an array-like of bools or 0s/1s where True represents a DST hour and False a non-DST hour, allows for distinguishing more than one DST transition (e.g., if you have multiple records in a database each with their own DST transition). Or passing ‘NaT’ will fill in transition times with not-a-time values. These methods are available in the `DatetimeIndex` constructor as well as `tz_localize`.

```

In [415]: rng_hourly_dst = np.array([1, 1, 0, 0, 0])

In [416]: rng_hourly.tz_localize('US/Eastern', ambiguous=rng_hourly_dst).tolist()

```

(continues on next page)

(continued from previous page)

```

Out[416]:
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0400', tz='US/Eastern'),
 Timestamp('2011-11-06 01:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]

In [417]: rng_hourly.tz_localize('US/Eastern', ambiguous='NaT').tolist()
///////////////////////////////////////////////////
↳
[Timestamp('2011-11-06 00:00:00-0400', tz='US/Eastern'),
 NaT,
 NaT,
 Timestamp('2011-11-06 02:00:00-0500', tz='US/Eastern'),
 Timestamp('2011-11-06 03:00:00-0500', tz='US/Eastern')]

In [418]: didx = pd.DatetimeIndex(start='2014-08-01 09:00', freq='H', periods=10, tz=
↳ 'US/Eastern')

In [419]: didx
Out[419]:
DatetimeIndex(['2014-08-01 09:00:00-04:00', '2014-08-01 10:00:00-04:00',
               '2014-08-01 11:00:00-04:00', '2014-08-01 12:00:00-04:00',
               '2014-08-01 13:00:00-04:00', '2014-08-01 14:00:00-04:00',
               '2014-08-01 15:00:00-04:00', '2014-08-01 16:00:00-04:00',
               '2014-08-01 17:00:00-04:00', '2014-08-01 18:00:00-04:00'],
              dtype='datetime64[ns, US/Eastern]', freq='H')

In [420]: didx.tz_localize(None)
///////////////////////////////////////////////////
↳
DatetimeIndex(['2014-08-01 09:00:00', '2014-08-01 10:00:00',
               '2014-08-01 11:00:00', '2014-08-01 12:00:00',
               '2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00'],
              dtype='datetime64[ns]', freq='H')

In [421]: didx.tz_convert(None)
///////////////////////////////////////////////////
↳
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00',
               '2014-08-01 19:00:00', '2014-08-01 20:00:00',
               '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
              dtype='datetime64[ns]', freq='H')

# tz_convert(None) is identical with tz_convert('UTC').tz_localize(None)
In [422]: didx.tz_convert('UCT').tz_localize(None)
///////////////////////////////////////////////////
↳
DatetimeIndex(['2014-08-01 13:00:00', '2014-08-01 14:00:00',
               '2014-08-01 15:00:00', '2014-08-01 16:00:00',
               '2014-08-01 17:00:00', '2014-08-01 18:00:00',
               '2014-08-01 19:00:00', '2014-08-01 20:00:00',
               '2014-08-01 21:00:00', '2014-08-01 22:00:00'],
              dtype='datetime64[ns]', freq='H')

```

(continues on next page)

(continued from previous page)

```
dtype='datetime64[ns]', freq='H')
```

### 19.14.3 TZ Aware Dtypes

Series/DatetimeIndex with a timezone **naive** value are represented with a dtype of `datetime64[ns]`.

```
In [423]: s_naive = pd.Series(pd.date_range('20130101', periods=3))

In [424]: s_naive
Out[424]:
0    2013-01-01
1    2013-01-02
2    2013-01-03
dtype: datetime64[ns]
```

Series/DatetimeIndex with a timezone **aware** value are represented with a dtype of `datetime64[ns, tz]`.

```
In [425]: s_aware = pd.Series(pd.date_range('20130101', periods=3, tz='US/Eastern'))

In [426]: s_aware
Out[426]:
0    2013-01-01 00:00:00-05:00
1    2013-01-02 00:00:00-05:00
2    2013-01-03 00:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

Both of these Series can be manipulated via the `.dt` accessor, see [here](#).

For example, to localize and convert a naive stamp to timezone aware.

```
In [427]: s_naive.dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[427]:
0    2012-12-31 19:00:00-05:00
1    2013-01-01 19:00:00-05:00
2    2013-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

Further more you can `.astype(...)` timezone aware (and naive). This operation is effectively a localize AND convert on a naive stamp, and a convert on an aware stamp.

```
# localize and convert a naive timezone
In [428]: s_naive.astype('datetime64[ns, US/Eastern]')
Out[428]:
0    2012-12-31 19:00:00-05:00
1    2013-01-01 19:00:00-05:00
2    2013-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]

# make an aware tz naive
In [429]: s_aware.astype('datetime64[ns]')
Out[429]:
0    2013-01-01 05:00:00
1    2013-01-02 05:00:00
```

(continues on next page)

(continued from previous page)

```

2    2013-01-03 05:00:00
dtype: datetime64[ns]

# convert to a new timezone
In [430]: s_aware.astype('datetime64[ns, CET]')
///////////////////////////////////////////////////
↪
0    2013-01-01 06:00:00+01:00
1    2013-01-02 06:00:00+01:00
2    2013-01-03 06:00:00+01:00
dtype: datetime64[ns, CET]

```

**Note:** Using the `.values` accessor on a `Series`, returns an NumPy array of the data. These values are converted to UTC, as NumPy does not currently support timezones (even though it is *printing* in the local timezone!).

```

In [431]: s_naive.values
Out[431]:
array(['2013-01-01T00:00:00.000000000', '2013-01-02T00:00:00.000000000',
      '2013-01-03T00:00:00.000000000'], dtype='datetime64[ns]')

In [432]: s_aware.values
///////////////////////////////////////////////////
↪
array(['2013-01-01T05:00:00.000000000', '2013-01-02T05:00:00.000000000',
      '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')

```

Further note that once converted to a NumPy array these would lose the tz tenor.

```

In [433]: pd.Series(s_aware.values)
Out[433]:
0    2013-01-01 05:00:00
1    2013-01-02 05:00:00
2    2013-01-03 05:00:00
dtype: datetime64[ns]

```

However, these can be easily converted:

```

In [434]: pd.Series(s_aware.values).dt.tz_localize('UTC').dt.tz_convert('US/Eastern')
Out[434]:
0    2013-01-01 00:00:00-05:00
1    2013-01-02 00:00:00-05:00
2    2013-01-03 00:00:00-05:00
dtype: datetime64[ns, US/Eastern]

```





## TIME DELTAS

Timedeltas are differences in times, expressed in difference units, e.g. days, hours, minutes, seconds. They can be both positive and negative.

Timedelta is a subclass of `datetime.timedelta`, and behaves in a similar manner, but allows compatibility with `np.timedelta64` types as well as a host of custom representation, parsing, and attributes.

### 20.1 Parsing

You can construct a `Timedelta` scalar through various arguments:

```
# strings
In [1]: pd.Timedelta('1 days')
Out[1]: Timedelta('1 days 00:00:00')

In [2]: pd.Timedelta('1 days 00:00:00')
Out[2]: Timedelta('1 days 00:00:00')

In [3]: pd.Timedelta('1 days 2 hours')
Out[3]: Timedelta('1 days 02:00:00')

In [4]: pd.Timedelta('-1 days 2 min 3us')
Out[4]: Timedelta('-2 days +23:57:59.999997')

# like datetime.timedelta
# note: these MUST be specified as keyword arguments
In [5]: pd.Timedelta(days=1, seconds=1)
Out[5]: Timedelta('1 days 00:00:01')

# integers with a unit
In [6]: pd.Timedelta(1, unit='d')
Out[6]: Timedelta('1 days 00:00:00')

# from a datetime.timedelta/np.timedelta64
In [7]: pd.Timedelta(datetime.timedelta(days=1, seconds=1))
Out[7]: Timedelta('1 days 00:00:01')

In [8]: pd.Timedelta(np.timedelta64(1, 'ms'))
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪Timedelta('0 days 00:00:00.001000')

# negative Timedeltas have this string repr
# to be more consistent with datetime.timedelta conventions
In [9]: pd.Timedelta('-1us')
////////////////////////////////////
↪Timedelta('-1 days +23:59:59.999999')

# a NaT
In [10]: pd.Timedelta('nan')
////////////////////////////////////
↪NaT

In [11]: pd.Timedelta('nat')
////////////////////////////////////
↪NaT

# ISO 8601 Duration strings
In [12]: pd.Timedelta('P0DT0H1M0S')
////////////////////////////////////
↪Timedelta('0 days 00:01:00')

In [13]: pd.Timedelta('P0DT0H0M0.000000123S')
////////////////////////////////////
↪Timedelta('0 days 00:00:00.000000')

```

New in version 0.23.0: Added constructor for [ISO 8601 Duration strings](#)

*DateOffsets* (Day, Hour, Minute, Second, Milli, Micro, Nano) can also be used in construction.

```

In [14]: pd.Timedelta(Second(2))
Out [14]: Timedelta('0 days 00:00:02')

```

Further, operations among the scalars yield another scalar `Timedelta`.

```

In [15]: pd.Timedelta(Day(2)) + pd.Timedelta(Second(2)) + pd.Timedelta('00:00:00.
↪000123')
Out [15]: Timedelta('2 days 00:00:02.000123')

```

### 20.1.1 to\_timedelta

Using the top-level `pd.to_timedelta`, you can convert a scalar, array, list, or `Series` from a recognized `timedelta` format / value into a `Timedelta` type. It will construct `Series` if the input is a `Series`, a scalar if the input is scalar-like, otherwise it will output a `TimedeltaIndex`.

You can parse a single string to a `Timedelta`:

```

In [16]: pd.to_timedelta('1 days 06:05:01.00003')
Out [16]: Timedelta('1 days 06:05:01.000030')

In [17]: pd.to_timedelta('15.5us')
////////////////////////////////////Out [17]: Timedelta('0 days 00:00:00.
↪000015')

```

or a list/array of strings:

```
In [18]: pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
Out[18]: TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT],
↳dtype='timedelta64[ns]', freq=None)
```

The unit keyword argument specifies the unit of the Timedelta:

```
In [19]: pd.to_timedelta(np.arange(5), unit='s')
Out[19]: TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02', '00:00:03', '00:00:04'],
↳dtype='timedelta64[ns]', freq=None)

In [20]: pd.to_timedelta(np.arange(5), unit='d')
////////////////////////////////////Out[20]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
↳'timedelta64[ns]', freq=None)
```

### 20.1.2 Timedelta limitations

Pandas represents Timedeltas in nanosecond resolution using 64 bit integers. As such, the 64 bit integer limits determine the Timedelta limits.

```
In [21]: pd.Timedelta.min
Out[21]: Timedelta('-106752 days +00:12:43.145224')

In [22]: pd.Timedelta.max
////////////////////////////////////Out[22]: Timedelta('106751 days
↳23:47:16.854775')
```

## 20.2 Operations

You can operate on Series/DataFrames and construct timedelta64[ns] Series through subtraction operations on datetime64[ns] Series, or Timestamps.

```
In [23]: s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))

In [24]: td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])

In [25]: df = pd.DataFrame(dict(A = s, B = td))

In [26]: df
Out[26]:
      A      B
0 2012-01-01 0 days
1 2012-01-02 1 days
2 2012-01-03 2 days

In [27]: df['C'] = df['A'] + df['B']

In [28]: df
Out[28]:
      A      B      C
0 2012-01-01 0 days 2012-01-01
1 2012-01-02 1 days 2012-01-03
2 2012-01-03 2 days 2012-01-05
```

(continues on next page)

(continued from previous page)

**In [29]:** df.dtypes

```

////////////////////////////////////
↪
A      datetime64[ns]
B      timedelta64[ns]
C      datetime64[ns]
dtype: object

```

**In [30]:** s - s.max()

```

////////////////////////////////////
↪
0    -2 days
1    -1 days
2     0 days
dtype: timedelta64[ns]

```

**In [31]:** s - datetime.datetime(2011, 1, 1, 3, 5)

```

////////////////////////////////////
↪
0    364 days 20:55:00
1    365 days 20:55:00
2    366 days 20:55:00
dtype: timedelta64[ns]

```

**In [32]:** s + datetime.timedelta(minutes=5)

```

////////////////////////////////////
↪
0    2012-01-01 00:05:00
1    2012-01-02 00:05:00
2    2012-01-03 00:05:00
dtype: datetime64[ns]

```

**In [33]:** s + Minute(5)

```

////////////////////////////////////
↪
0    2012-01-01 00:05:00
1    2012-01-02 00:05:00
2    2012-01-03 00:05:00
dtype: datetime64[ns]

```

**In [34]:** s + Minute(5) + Milli(5)

```

////////////////////////////////////
↪
0    2012-01-01 00:05:00.005
1    2012-01-02 00:05:00.005
2    2012-01-03 00:05:00.005
dtype: datetime64[ns]

```

Operations with scalars from a `timedelta64[ns]` series:**In [35]:** y = s - s[0]**In [36]:** y**Out[36]:**

```

0    0 days
1    1 days

```

(continues on next page)

(continued from previous page)

```
2    2 days
dtype: timedelta64[ns]
```

Series of timedeltas with NaT values are supported:

```
In [37]: y = s - s.shift()

In [38]: y
Out[38]:
0      NaT
1    1 days
2    1 days
dtype: timedelta64[ns]
```

Elements can be set to NaT using `np.nan` analogously to datetimes:

```
In [39]: y[1] = np.nan

In [40]: y
Out[40]:
0      NaT
1      NaT
2    1 days
dtype: timedelta64[ns]
```

Operands can also appear in a reversed order (a singular object operated with a Series):

```
In [41]: s.max() - s
Out[41]:
0    2 days
1    1 days
2    0 days
dtype: timedelta64[ns]

In [42]: datetime.datetime(2011, 1, 1, 3, 5) - s
Out[42]:
0   -365 days +03:05:00
1   -366 days +03:05:00
2   -367 days +03:05:00
dtype: timedelta64[ns]

In [43]: datetime.datetime(2012, 1, 1, 0, 5) + s
Out[43]:
0   2012-01-01 00:05:00
1   2012-01-02 00:05:00
2   2012-01-03 00:05:00
dtype: datetime64[ns]
```

`min`, `max` and the corresponding `idxmin`, `idxmax` operations are supported on frames:

```
In [44]: A = s - pd.Timestamp('20120101') - pd.Timedelta('00:05:05')

In [45]: B = s - pd.Series(pd.date_range('2012-1-2', periods=3, freq='D'))

In [46]: df = pd.DataFrame(dict(A=A, B=B))
```

(continues on next page)

(continued from previous page)

```
In [47]: df
Out[47]:
```

|   | A                 | B       |
|---|-------------------|---------|
| 0 | -1 days +23:54:55 | -1 days |
| 1 | 0 days 23:54:55   | -1 days |
| 2 | 1 days 23:54:55   | -1 days |

```
In [48]: df.min()
Out[48]:
A    -1 days +23:54:55
B    -1 days +00:00:00
dtype: timedelta64[ns]
```

```
In [49]: df.min(axis=1)
Out[49]:
0    -1 days
1    -1 days
2    -1 days
dtype: timedelta64[ns]
```

```
In [50]: df.idxmin()
Out[50]:
A    0
B    0
dtype: int64
```

```
In [51]: df.idxmax()
Out[51]:
A    2
B    0
dtype: int64
```

min, max, idxmin, idxmax operations are supported on Series as well. A scalar result will be a Timedelta.

```
In [52]: df.min().max()
Out[52]: Timedelta('-1 days +23:54:55')

In [53]: df.min(axis=1).min()
Out[53]: Timedelta('-1 days +00:00:00')
```

```
In [54]: df.min().idxmax()
Out[54]: 'A'
```

```
In [55]: df.min(axis=1).idxmin()
Out[55]: 0
```

You can fillna on timedeltas. Integers will be interpreted as seconds. You can pass a timedelta to get a particular value.

```
In [56]: y.fillna(0)
Out[56]:
```

(continues on next page)

(continued from previous page)

```

0    0 days
1    0 days
2    1 days
dtype: timedelta64[ns]

In [57]: y.fillna(10)
Out[57]:
0    0 days 00:00:10
1    0 days 00:00:10
2    1 days 00:00:00
dtype: timedelta64[ns]

In [58]: y.fillna(pd.Timedelta('-1 days, 00:00:05'))
Out[58]:
0    -1 days +00:00:05
1    -1 days +00:00:05
2     1 days 00:00:00
dtype: timedelta64[ns]

```

You can also negate, multiply and use abs on Timedeltas:

```

In [59]: td1 = pd.Timedelta('-1 days 2 hours 3 seconds')

In [60]: td1
Out[60]: Timedelta('-2 days +21:59:57')

In [61]: -1 * td1
Out[61]: Timedelta('1 days 02:00:03')

In [62]: -td1
Out[62]: Timedelta('1 days 02:00:03')

In [63]: abs(td1)
Out[63]: Timedelta('1 days 02:00:03')

```

## 20.3 Reductions

Numeric reduction operation for `timedelta64[ns]` will return `Timedelta` objects. As usual `NaT` are skipped during evaluation.

```

In [64]: y2 = pd.Series(pd.to_timedelta(['-1 days +00:00:05', 'nat', '-1 days_
Out[64]:
0    -1 days +00:00:05
1         NaT
2    -1 days +00:00:05
3     1 days 00:00:00
dtype: timedelta64[ns]

```

(continues on next page)

(continued from previous page)

[illegible][illegible]

## 20.4 Frequency Conversion

Timedelta Series, TimedeltaIndex, and Timedelta scalars can be converted to other ‘frequencies’ by dividing by another timedelta, or by astyping to a specific timedelta type. These operations yield Series and propagate NaT -> nan. Note that division by the NumPy scalar is true division, while astyping is equivalent of floor division.

```
In [70]: td = pd.Series(pd.date_range('20130101', periods=4)) - \
.....:     pd.Series(pd.date_range('20121201', periods=4))
.....:
```

```
In [71]: td[2] += datetime.timedelta(minutes=5, seconds=3)
```

```
In [72]: td[3] = np.nan
```

```
In [73]: td
```

Out [73]:

```
0    31 days 00:00:00
1    31 days 00:00:00
2    31 days 00:05:03
3              NaT
dtype: timedelta64[ns]
```

```
# to days
```

```
In [74]: td / np.timedelta64(1, 'D')
```

```
0      31.000000
1      31.000000
2      31.003507
3           NaN
dtype: float64
```

```
In [75]: td.astype('timedelta64[D]')
```

|   |      |
|---|------|
|   |      |
| ↔ |      |
| 0 | 31.0 |
| 1 | 31.0 |
| 2 | 31.0 |

(continues on next page)



```
# to seconds
```

```
In [77]: td.astype('timedelta64[s]')
```

```
# to months (these are constant months)
```

Dividing or multiplying a `timedelta64[ns]` Series by an integer or integer Series yields another `timedelta64[ns]` dtypes Series.

```
In [80]: td * pd.Series([1, 2, 3, 4])
```

Rounded division (floor-division) of a `timedelta64[ns]` Series by a scalar `Timedelta` gives a series of integers.

---

(continues on next page)

(continued from previous page)

```

0    9.0
1    9.0
2    9.0
3    NaN
dtype: float64

In [82]: pd.Timedelta(days=3, hours=4) // td
Out[82]:
0    0.0
1    0.0
2    0.0
3    NaN
dtype: float64

```

The mod (%) and divmod operations are defined for Timedelta when operating with another timedelta-like or with a numeric argument.

```

In [83]: pd.Timedelta(hours=37) % datetime.timedelta(hours=2)
Out[83]: Timedelta('0 days 01:00:00')

# divmod against a timedelta-like returns a pair (int, Timedelta)
In [84]: divmod(datetime.timedelta(hours=2), pd.Timedelta(minutes=11))
Out[84]: (10, Timedelta('0 days 00:10:00'))

# divmod against a numeric returns a pair (Timedelta, Timedelta)
In [85]: divmod(pd.Timedelta(hours=25), 864000000000000)
Out[85]: (Timedelta('0 days 00:00:00.000000'), Timedelta('0 days 01:00:00'))

```

## 20.5 Attributes

You can access various components of the Timedelta or TimedeltaIndex directly using the attributes `days`, `seconds`, `microseconds`, `nanoseconds`. These are identical to the values returned by `datetime.timedelta`, in that, for example, the `.seconds` attribute represents the number of seconds  $\geq 0$  and  $< 1$  day. These are signed according to whether the Timedelta is signed.

These operations can also be directly accessed via the `.dt` property of the Series as well.

**Note:** Note that the attributes are NOT the displayed values of the Timedelta. Use `.components` to retrieve the displayed values.

For a Series:

```

In [86]: td.dt.days
Out[86]:
0    31.0
1    31.0
2    31.0
3    NaN
dtype: float64

In [87]: td.dt.seconds

```

(continues on next page)

```
Out[87]:
0      0.0
1      0.0
2     303.0
3      NaN
dtype: float64
```

Using `TimedeltaIndex` you can pass string-like, `Timedelta`, `timedelta`, or `np.timedelta64` objects. Passing `np.nan/pd.NaT/nat` will represent missing values.

```
In [95]: pd.TimedeltaIndex(['1 days', '1 days, 00:00:05',
.....:                      np.timedelta64(2, 'D'), datetime.timedelta(days=2,
↳seconds=2)])
.....:
Out[95]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:00:05', '2 days 00:00:00',
               '2 days 00:00:02'],
               dtype='timedelta64[ns]', freq=None)
```

### 20.6.1 Generating Ranges of Time Deltas

Similar to `date_range()`, you can construct regular ranges of a `TimedeltaIndex` using `timedelta_range()`. The default frequency for `timedelta_range` is calendar day:

```
In [96]: pd.timedelta_range(start='1 days', periods=5)
Out[96]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↳ 'timedelta64[ns]', freq='D')
```

Various combinations of `start`, `end`, and `periods` can be used with `timedelta_range`:

```
In [97]: pd.timedelta_range(start='1 days', end='5 days')
Out[97]: TimedeltaIndex(['1 days', '2 days', '3 days', '4 days', '5 days'], dtype=
↳ 'timedelta64[ns]', freq='D')

In [98]: pd.timedelta_range(end='10 days', periods=4)
\\////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↳ TimedeltaIndex(['7 days', '8 days', '9 days', '10 days'], dtype='timedelta64[ns]
↳ freq='D')
```

The `freq` parameter can be passed a variety of *frequency aliases*:

```
In [99]: pd.timedelta_range(start='1 days', end='2 days', freq='30T')
Out[99]:
TimedeltaIndex(['1 days 00:00:00', '1 days 00:30:00', '1 days 01:00:00',
                '1 days 01:30:00', '1 days 02:00:00', '1 days 02:30:00',
                '1 days 03:00:00', '1 days 03:30:00', '1 days 04:00:00',
                '1 days 04:30:00', '1 days 05:00:00', '1 days 05:30:00',
                '1 days 06:00:00', '1 days 06:30:00', '1 days 07:00:00',
                '1 days 07:30:00', '1 days 08:00:00', '1 days 08:30:00',
                '1 days 09:00:00', '1 days 09:30:00', '1 days 10:00:00',
                '1 days 10:30:00', '1 days 11:00:00', '1 days 11:30:00',
                '1 days 12:00:00', '1 days 12:30:00', '1 days 13:00:00',
                '1 days 13:30:00', '1 days 14:00:00', '1 days 14:30:00',
                '1 days 15:00:00', '1 days 15:30:00', '1 days 16:00:00',
                '1 days 16:30:00', '1 days 17:00:00', '1 days 17:30:00',
                '1 days 18:00:00', '1 days 18:30:00', '1 days 19:00:00',
                '1 days 19:30:00', '1 days 20:00:00', '1 days 20:30:00',
                '1 days 21:00:00', '1 days 21:30:00', '1 days 22:00:00',
                '1 days 22:30:00', '1 days 23:00:00', '1 days 23:30:00',
                '2 days 00:00:00'],
                dtype='timedelta64[ns]', freq='30T')

In [100]: pd.timedelta_range(start='1 days', periods=5, freq='2D5H')
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
TimedeltaIndex(['1 days 00:00:00', '3 days 05:00:00', '5 days 10:00:00',
               '7 days 15:00:00', '9 days 20:00:00'],
              dtype='timedelta64[ns]', freq='53H')

```

New in version 0.23.0.

Specifying start, end, and periods will generate a range of evenly spaced timesteps from start to end inclusively, with periods number of elements in the resulting `TimedeltaIndex`:

```

In [101]: pd.timedelta_range('0 days', '4 days', periods=5)
Out[101]: TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'], dtype=
↪ 'timedelta64[ns]', freq=None)

In [102]: pd.timedelta_range('0 days', '4 days', periods=10)
////////////////////////////////////
↪
TimedeltaIndex(['0 days 00:00:00', '0 days 10:40:00', '0 days 21:20:00',
               '1 days 08:00:00', '1 days 18:40:00', '2 days 05:20:00',
               '2 days 16:00:00', '3 days 02:40:00', '3 days 13:20:00',
               '4 days 00:00:00'],
              dtype='timedelta64[ns]', freq=None)

```

## 20.6.2 Using the `TimedeltaIndex`

Similarly to other of the datetime-like indices, `DatetimeIndex` and `PeriodIndex`, you can use `TimedeltaIndex` as the index of pandas objects.

```

In [103]: s = pd.Series(np.arange(100),
.....:                  index=pd.timedelta_range('1 days', periods=100, freq='h'))
.....:

In [104]: s
Out[104]:
1 days 00:00:00    0
1 days 01:00:00    1
1 days 02:00:00    2
1 days 03:00:00    3
1 days 04:00:00    4
1 days 05:00:00    5
1 days 06:00:00    6
..
4 days 21:00:00   93
4 days 22:00:00   94
4 days 23:00:00   95
5 days 00:00:00   96
5 days 01:00:00   97
5 days 02:00:00   98
5 days 03:00:00   99
Freq: H, Length: 100, dtype: int64

```

Selections work similarly, with coercion on string-likes and slices:

```
In [105]: s['1 day':'2 day']
Out[105]:
1 days 00:00:00    0
1 days 01:00:00    1
1 days 02:00:00    2
1 days 03:00:00    3
1 days 04:00:00    4
1 days 05:00:00    5
1 days 06:00:00    6
..
2 days 17:00:00   41
2 days 18:00:00   42
2 days 19:00:00   43
2 days 20:00:00   44
2 days 21:00:00   45
2 days 22:00:00   46
2 days 23:00:00   47
Freq: H, Length: 48, dtype: int64
```

```
In [106]: s['1 day 01:00:00']
```

```

////////////////////////////////////
↪1
```

```
In [107]: s[pd.Timedelta('1 day 1h')]
```

```

////////////////////////////////////
↪1
```

Furthermore you can use partial string selection and the range will be inferred:

```
In [108]: s['1 day':'1 day 5 hours']
Out[108]:
1 days 00:00:00    0
1 days 01:00:00    1
1 days 02:00:00    2
1 days 03:00:00    3
1 days 04:00:00    4
1 days 05:00:00    5
Freq: H, dtype: int64
```

### 20.6.3 Operations

Finally, the combination of `TimedeltaIndex` with `DatetimeIndex` allow certain combination operations that are NaT preserving:

```
In [109]: tdi = pd.TimedeltaIndex(['1 days', pd.NaT, '2 days'])

In [110]: tdi.tolist()
Out[110]: [Timedelta('1 days 00:00:00'), NaT, Timedelta('2 days 00:00:00')]

In [111]: dti = pd.date_range('20130101', periods=3)

In [112]: dti.tolist()
Out[112]: [Timestamp('2013-01-01 00:00:00', freq='D'),
Timestamp('2013-01-02 00:00:00', freq='D'),
Timestamp('2013-01-03 00:00:00', freq='D')]
```

(continues on next page)

(continued from previous page)

```

In [113]: (dti + tdi).tolist()
\\Out[113]: [Timestamp('2013-01-02 00:00:00'), NaT, Timestamp('2013-01-05 00:00:00')]

In [114]: (dti - tdi).tolist()
\\Out[114]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2013-01-01 00:00:00')]

```

## 20.6.4 Conversions

Similarly to frequency conversion on a Series above, you can convert these indices to yield another Index.

```

In [115]: tdi / np.timedelta64(1, 's')
Out[115]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

In [116]: tdi.astype('timedelta64[s]')
\\Out[116]: Float64Index([86400.0, nan, 172800.0], dtype='float64')

```

Scalars type ops work as well. These can potentially return a *different* type of index.

```

# adding or timedelta and date -> datelike
In [117]: tdi + pd.Timestamp('20130101')
Out[117]: DatetimeIndex(['2013-01-02', 'NaT', '2013-01-03'], dtype='datetime64[ns]', freq=None)

# subtraction of a date and a timedelta -> datelike
# note that trying to subtract a date from a Timedelta will raise an exception
In [118]: (pd.Timestamp('20130101') - tdi).tolist()
\\Out[118]: [Timestamp('2012-12-31 00:00:00'), NaT, Timestamp('2012-12-30 00:00:00')]

# timedelta + timedelta -> timedelta
In [119]: tdi + pd.Timedelta('10 days')
\\Out[119]: TimedeltaIndex(['11 days', 'NaT', '12 days'], dtype='timedelta64[ns]', freq=None)

# division can result in a Timedelta if the divisor is an integer
In [120]: tdi / 2
\\Out[120]: TimedeltaIndex(['0 days 12:00:00', 'NaT', '1 days 00:00:00'], dtype='timedelta64[ns]', freq=None)

# or a Float64Index if the divisor is a Timedelta
In [121]: tdi / tdi[0]
\\Out[121]: Float64Index([1.0, nan, 2.0], dtype='float64')

```

## 20.7 Resampling

Similar to *timeseries resampling*, we can resample with a TimedeltaIndex.

```
In [122]: s.resample('D').mean()
Out[122]:
1 days    11.5
2 days    35.5
3 days    59.5
4 days    83.5
5 days    97.5
Freq: D, dtype: float64
```



## CATEGORICAL DATA

This is an introduction to pandas categorical data type, including a short comparison with R's `factor`.

*Categoricals* are a pandas data type corresponding to categorical variables in statistics. A categorical variable takes on a limited, and usually fixed, number of possible values (*categories*; *levels* in R). Examples are gender, social class, blood type, country affiliation, observation time or rating via Likert scales.

In contrast to statistical categorical variables, categorical data might have an order (e.g. 'strongly agree' vs 'agree' or 'first observation' vs. 'second observation'), but numerical operations (additions, divisions, ...) are not possible.

All values of categorical data are either in *categories* or *np.nan*. Order is defined by the order of *categories*, not lexical order of the values. Internally, the data structure consists of a *categories* array and an integer array of *codes* which point to the real value in the *categories* array.

The categorical data type is useful in the following cases:

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory, see [here](#).
- The lexical order of a variable is not the same as the logical order ("one", "two", "three"). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order, see [here](#).
- As a signal to other Python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

See also the [API docs on categoricals](#).

## 21.1 Object Creation

### 21.1.1 Series Creation

Categorical Series or columns in a DataFrame can be created in several ways:

By specifying `dtype="category"` when constructing a Series:

```
In [1]: s = pd.Series(["a", "b", "c", "a"], dtype="category")

In [2]: s
Out[2]:
0      a
1      b
2      c
3      a
```

(continues on next page)

(continued from previous page)

```
dtype: category
Categories (3, object): [a, b, c]
```

By converting an existing Series or column to a category dtype:

```
In [3]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})

In [4]: df["B"] = df["A"].astype('category')

In [5]: df
Out[5]:
   A B
0  a a
1  b b
2  c c
3  a a
```

By using special functions, such as `cut()`, which groups data into discrete bins. See the [example on tiling](#) in the docs.

```
In [6]: df = pd.DataFrame({'value': np.random.randint(0, 100, 20)})

In [7]: labels = ["{0} - {1}".format(i, i + 9) for i in range(0, 100, 10)]

In [8]: df['group'] = pd.cut(df.value, range(0, 105, 10), right=False, labels=labels)

In [9]: df.head(10)
Out[9]:
   value  group
0     65  60 - 69
1     49  40 - 49
2     56  50 - 59
3     43  40 - 49
4     43  40 - 49
5     91  90 - 99
6     32  30 - 39
7     87  80 - 89
8     36  30 - 39
9      8   0 - 9
```

By passing a `pandas.Categorical` object to a Series or assigning it to a DataFrame.

```
In [10]: raw_cat = pd.Categorical(["a", "b", "c", "a"], categories=["b", "c", "d"],
....:                             ordered=False)
....:

In [11]: s = pd.Series(raw_cat)

In [12]: s
Out[12]:
0    NaN
1     b
2     c
3    NaN
dtype: category
Categories (3, object): [b, c, d]
```

(continues on next page)

(continued from previous page)

```
In [13]: df = pd.DataFrame({"A": ["a", "b", "c", "a"]})
In [14]: df["B"] = raw_cat
In [15]: df
Out[15]:
   A    B
0  a  NaN
1  b    b
2  c    c
3  a  NaN
```

Categorical data has a specific category *dtype*:

```
In [16]: df.dtypes
Out[16]:
A      object
B    category
dtype: object
```

## 21.1.2 DataFrame Creation

Similar to the previous section where a single column was converted to categorical, all columns in a DataFrame can be batch converted to categorical either during or after construction.

This can be done during construction by specifying `dtype="category"` in the DataFrame constructor:

```
In [17]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')}, dtype="category")
In [18]: df.dtypes
Out[18]:
A    category
B    category
dtype: object
```

Note that the categories present in each column differ; the conversion is done column by column, so only labels present in a given column are categories:

```
In [19]: df['A']
Out[19]:
0    a
1    b
2    c
3    a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [20]: df['B']
Out[20]:
0    b
1    c
2    c
3    d
```

(continues on next page)

(continued from previous page)

```
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

New in version 0.23.0.

Analogously, all columns in an existing DataFrame can be batch converted using `DataFrame.astype()`:

```
In [21]: df = pd.DataFrame({'A': list('abca'), 'B': list('bccd')})
In [22]: df_cat = df.astype('category')
In [23]: df_cat.dtypes
Out[23]:
A      category
B      category
dtype: object
```

This conversion is likewise done column by column:

```
In [24]: df_cat['A']
Out[24]:
0      a
1      b
2      c
3      a
Name: A, dtype: category
Categories (3, object): [a, b, c]

In [25]: df_cat['B']
Out[25]:
0      b
1      c
2      c
3      d
Name: B, dtype: category
Categories (3, object): [b, c, d]
```

### 21.1.3 Controlling Behavior

In the examples above where we passed `dtype='category'`, we used the default behavior:

1. Categories are inferred from the data.
2. Categories are unordered.

To control those behaviors, instead of passing `'category'`, use an instance of `CategoricalDtype`.

```
In [26]: from pandas.api.types import CategoricalDtype
In [27]: s = pd.Series(["a", "b", "c", "a"])
In [28]: cat_type = CategoricalDtype(categories=["b", "c", "d"],
.....:                                     ordered=True)
.....:
```

(continues on next page)



### 21.1.4 Regaining Original Data

To get back to the original Series or NumPy array, use `Series.astype(original_dtype)` or `np.asarray(categorical)`:

```
In [38]: s = pd.Series(["a", "b", "c", "a"])

In [39]: s
Out[39]:
0    a
1    b
2    c
3    a
dtype: object

In [40]: s2 = s.astype('category')

In [41]: s2
Out[41]:
0    a
1    b
2    c
3    a
dtype: category
Categories (3, object): [a, b, c]

In [42]: s2.astype(str)
Out[42]:
0    a
1    b
2    c
3    a
dtype: object

In [43]: np.asarray(s2)
array(['a', 'b', 'c', 'a'], dtype=object)
```

---

**Note:** In contrast to R's *factor* function, categorical data is not converting input values to strings; categories will end up the same data type as the original values.

---

---

**Note:** In contrast to R's *factor* function, there is currently no way to assign/change labels at creation time. Use *categories* to change the categories after creation time.

---

## 21.2 CategoricalDtype

Changed in version 0.21.0.

A categorical's type is fully described by

1. `categories`: a sequence of unique values and no missing values

2. `ordered`: a boolean

This information can be stored in a `CategoricalDtype`. The `categories` argument is optional, which implies that the actual categories should be inferred from whatever is present in the data when the `pandas.Categorical` is created. The categories are assumed to be unordered by default.

```
In [44]: from pandas.api.types import CategoricalDtype

In [45]: CategoricalDtype(['a', 'b', 'c'])
Out[45]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=None)

In [46]: CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[46]: CategoricalDtype(categories=['a', 'b', 'c'], ordered=True)

In [47]: CategoricalDtype()
Out[47]: CategoricalDtype(categories=None, ordered=None)
```

A `CategoricalDtype` can be used in any place pandas expects a `dtype`. For example `pandas.read_csv()`, `pandas.DataFrame.astype()`, or in the Series constructor.

**Note:** As a convenience, you can use the string `'category'` in place of a `CategoricalDtype` when you want the default behavior of the categories being unordered, and equal to the set values present in the array. In other words, `dtype='category'` is equivalent to `dtype=CategoricalDtype()`.

### 21.2.1 Equality Semantics

Two instances of `CategoricalDtype` compare equal whenever they have the same categories and order. When comparing two unordered categoricals, the order of the categories is not considered.

```
In [48]: c1 = CategoricalDtype(['a', 'b', 'c'], ordered=False)

# Equal, since order is not considered when ordered=False
In [49]: c1 == CategoricalDtype(['b', 'c', 'a'], ordered=False)
Out[49]: True

# Unequal, since the second CategoricalDtype is ordered
In [50]: c1 == CategoricalDtype(['a', 'b', 'c'], ordered=True)
Out[50]: False
```

All instances of `CategoricalDtype` compare equal to the string `'category'`.

```
In [51]: c1 == 'category'
Out[51]: True
```

**Warning:** Since `dtype='category'` is essentially `CategoricalDtype(None, False)`, and since all instances `CategoricalDtype` compare equal to `'category'`, all instances of `CategoricalDtype` compare equal to a `CategoricalDtype(None, False)`, regardless of categories or ordered.

## 21.3 Description

Using `describe()` on categorical data will produce similar output to a Series or DataFrame of type string.

```
In [52]: cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a", "c"])

In [53]: df = pd.DataFrame({"cat":cat, "s":["a", "c", "c", np.nan]})

In [54]: df.describe()
Out[54]:
```

|        | cat | s |
|--------|-----|---|
| count  | 3   | 3 |
| unique | 2   | 2 |
| top    | c   | c |
| freq   | 2   | 2 |

```
In [55]: df["cat"].describe()
Out[55]:
```

|        |   |
|--------|---|
| count  | 3 |
| unique | 2 |
| top    | c |
| freq   | 2 |

```
Name: cat, dtype: object
```

## 21.4 Working with categories

Categorical data has a *categories* and a *ordered* property, which list their possible values and whether the ordering matters or not. These properties are exposed as `s.cat.categories` and `s.cat.ordered`. If you don't manually specify categories and ordering, they are inferred from the passed arguments.

```
In [56]: s = pd.Series(["a","b","c","a"], dtype="category")

In [57]: s.cat.categories
Out[57]: Index(['a', 'b', 'c'], dtype='object')

In [58]: s.cat.ordered
Out[58]: False
```

It's also possible to pass in the categories in a specific order:

```
In [59]: s = pd.Series(pd.Categorical(["a", "b", "c", "a"], categories=["c", "b", "a"]))
In [60]: s.cat.categories
Out[60]: Index(['c', 'b', 'a'], dtype='object')

In [61]: s.cat.ordered
Out[61]: False
```

**Note:** New categorical data are **not** automatically ordered. You must explicitly pass `ordered=True` to indicate an ordered `Categorical`.



```

////////////////////////////////////////
↪
[b, a, c]
Categories (3, object): [b, a, c]

```

```
\\Out[64]:
```

```
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]
```

1019

(continued from previous page)

```
In [70]: s.cat.rename_categories([1,2,3])
```

```

////////////////////////////////////
↪
0      1
1      2
2      3
3      1
dtype: category
Categories (3, int64): [1, 2, 3]
```

```
In [71]: s
```

```

////////////////////////////////////
↪
0      Group a
1      Group b
2      Group c
3      Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]
```

```
# You can also pass a dict-like object to map the renaming
```

```
In [72]: s.cat.rename_categories({1: 'x', 2: 'y', 3: 'z'})
```

```

////////////////////////////////////
↪
0      Group a
1      Group b
2      Group c
3      Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]
```

```
In [73]: s
```

```

////////////////////////////////////
↪
0      Group a
1      Group b
2      Group c
3      Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]
```

---

**Note:** In contrast to R's *factor*, categorical data can have categories of other types than string.

---



---

**Note:** Be aware that assigning new categories is an inplace operation, while most other operations under `Series.cat` per default return a new `Series` of dtype `category`.

---

Categories must be unique or a `ValueError` is raised:

```
In [74]: try:
....:     s.cat.categories = [1,1,1]
....: except ValueError as e:
....:     print("ValueError: " + str(e))
....:
```

(continues on next page)

(continued from previous page)

```
ValueError: Categorical categories must be unique
```

Categories must also not be NaN or a *ValueError* is raised:

```
In [75]: try:
.....:     s.cat.categories = [1,2,np.nan]
.....: except ValueError as e:
.....:     print("ValueError: " + str(e))
.....:
ValueError: Categorical categories cannot be null
```

## 21.4.2 Appending new categories

Appending categories can be done by using the `add_categories()` method:

```
In [76]: s = s.cat.add_categories([4])

In [77]: s.cat.categories
Out[77]: Index(['Group a', 'Group b', 'Group c', 4], dtype='object')

In [78]: s
Out[78]:
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (4, object): [Group a, Group b, Group c, 4]
```

## 21.4.3 Removing categories

Removing categories can be done by using the `remove_categories()` method. Values which are removed are replaced by `np.nan`:

```
In [79]: s = s.cat.remove_categories([4])

In [80]: s
Out[80]:
0    Group a
1    Group b
2    Group c
3    Group a
dtype: category
Categories (3, object): [Group a, Group b, Group c]
```

## 21.4.4 Removing unused categories

Removing unused categories can also be done:

```
In [81]: s = pd.Series(pd.Categorical(["a","b","a"], categories=["a","b","c","d"]))
```

(continues on next page)

(continued from previous page)

```

In [82]: s
Out[82]:
0      a
1      b
2      a
dtype: category
Categories (4, object): [a, b, c, d]

In [83]: s.cat.remove_unused_categories()
Out[83]:
0      a
1      b
2      a
dtype: category
Categories (2, object): [a, b]

```

### 21.4.5 Setting categories

If you want to do remove and add new categories in one step (which has some speed advantage), or simply set the categories to a predefined scale, use `set_categories()`.

```

In [84]: s = pd.Series(["one", "two", "four", "-"], dtype="category")

In [85]: s
Out[85]:
0      one
1      two
2      four
3      -
dtype: category
Categories (4, object): [-, four, one, two]

In [86]: s = s.cat.set_categories(["one", "two", "three", "four"])

In [87]: s
Out[87]:
0      one
1      two
2      four
3      NaN
dtype: category
Categories (4, object): [one, two, three, four]

```

**Note:** Be aware that `Categorical.set_categories()` cannot know whether some category is omitted intentionally or because it is misspelled or (under Python3) due to a type difference (e.g., NumPy S1 dtype and Python strings). This can result in surprising behaviour!

[illegible]

```
In [94]: s.cat.as_ordered()
Out[94]:
0      a
3      a
1      b
2      c
dtype: category
Categories (3, object): [a < b < c]

In [95]: s.cat.as_unordered()
Out[95]:
0      a
3      a
1      b
2      c
dtype: category
Categories (3, object): [a, b, c]
```

```
In [96]: s = pd.Series([1,2,3,1], dtype="category")
In [97]: s = s.cat.set_categories([2,3,1], ordered=True)
```

1023

(continued from previous page)

[illegible]

### 21.5.1 Reordering

Reordering the categories is possible via the `Categorical.reorder_categories()` and the `Categorical.set_categories()` methods. For `Categorical.reorder_categories()`, all old categories must be included in the new categories and no new categories are allowed. This will necessarily make the sort order the same as the categories order.

```
In [102]: s = pd.Series([1,2,3,1], dtype="category")

In [103]: s = s.cat.reorder_categories([2,3,1], ordered=True)

In [104]: s
Out[104]:
0    1
1    2
2    3
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]

In [105]: s.sort_values(inplace=True)

In [106]: s
Out[106]:
1    2
2    3
0    1
3    1
dtype: category
Categories (3, int64): [2 < 3 < 1]
```

(continues on next page)

(continued from previous page)

```
In [107]: s.min(), s.max()
\\Out[107]:
↪ (2, 1)
```

**Note:** Note the difference between assigning new categories and reordering the categories: the first renames categories and therefore the individual values in the `Series`, but if the first position was sorted last, the renamed value will still be sorted last. Reordering means that the way values are sorted is different afterwards, but not that individual values in the `Series` are changed.

**Note:** If the `Categorical` is not ordered, `Series.min()` and `Series.max()` will raise `TypeError`. Numeric operations like `+`, `-`, `*`, `/` and operations based on them (e.g. `Series.median()`, which would need to compute the mean between two values if the length of an array is even) do not work and raise a `TypeError`.

## 21.5.2 Multi Column Sorting

A categorical dtyped column will participate in a multi-column sort in a similar manner to other columns. The ordering of the categorical is determined by the categories of that column.

```
In [108]: dfs = pd.DataFrame({'A' : pd.Categorical(list('bbeebbaa'), categories=['e',
↪ 'a', 'b'], ordered=True),
.....:                      'B' : [1,2,1,2,2,1,2,1] })
.....:

In [109]: dfs.sort_values(by=['A', 'B'])
Out[109]:
   A  B
2  e  1
3  e  2
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
```

Reordering the categories changes a future sort.

```
In [110]: dfs['A'] = dfs['A'].cat.reorder_categories(['a', 'b', 'e'])

In [111]: dfs.sort_values(by=['A', 'B'])
Out[111]:
   A  B
7  a  1
6  a  2
0  b  1
5  b  1
1  b  2
4  b  2
2  e  1
3  e  2
```

## 21.6 Comparisons

Comparing categorical data with other objects is possible in three cases:

- Comparing equality (`==` and `!=`) to a list-like object (list, Series, array, ...) of the same length as the categorical data.
- All comparisons (`==`, `!=`, `>`, `>=`, `<`, and `<=`) of categorical data to another categorical Series, when `ordered==True` and the *categories* are the same.
- All comparisons of a categorical data to a scalar.

All other comparisons, especially “non-equality” comparisons of two categoricals with different categories or a categorical with any list-like object, will raise a `TypeError`.

**Note:** Any “non-equality” comparisons of categorical data with a `Series`, `np.array`, `list` or categorical data with different categories or ordering will raise a `TypeError` because custom categories ordering could be interpreted in two ways: one with taking into account the ordering and one without.

```
In [112]: cat = pd.Series([1,2,3]).astype(
.....:     CategoricalDtype([3, 2, 1], ordered=True)
.....: )
.....:

In [113]: cat_base = pd.Series([2,2,2]).astype(
.....:     CategoricalDtype([3, 2, 1], ordered=True)
.....: )
.....:

In [114]: cat_base2 = pd.Series([2,2,2]).astype(
.....:     CategoricalDtype(ordered=True)
.....: )
.....:

In [115]: cat
Out[115]:
0    1
1    2
2    3
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [116]: cat_base
Out[116]:
0    2
1    2
2    2
dtype: category
Categories (3, int64): [3 < 2 < 1]

In [117]: cat_base2
Out[117]:
0    2
1    2
```

(continues on next page)





```
In [124]: base = np.array([1,2,3])  
  
In [125]: try:  
.....:     cat > base  
.....: except TypeError as e:  
.....:     print("TypeError: " + str(e))  
.....:  
TypeError: Cannot compare a Categorical for op __gt__ with type <class 'numpy.ndarray'  
↳ '>'.  
If you want to compare values, use 'np.asarray(cat) <op> other'.  
  
In [126]: np.asarray(cat) > base  
/////////////////////////////////////  
↳ array([False, False, False], dtype=bool)
```

When you compare two unordered categoricals with the same categories, the order is not considered:

```
In [127]: c1 = pd.Categorical(['a', 'b'], categories=['a', 'b'], ordered=False)
In [128]: c2 = pd.Categorical(['a', 'b'], categories=['b', 'a'], ordered=False)
In [129]: c1 == c2
Out[129]: array([ True,  True], dtype=bool)
```

## 21.7 Operations

Apart from `Series.min()`, `Series.max()` and `Series.mode()`, the following operations are possible with categorical data:

Series methods like `Series.value_counts()` will use all categories, even if some categories are not present in the data:

```
In [130]: s = pd.Series(pd.Categorical(["a", "b", "c", "c"], categories=["c", "a", "b", "d"
↳ ""]))

In [131]: s.value_counts()
Out[131]:
c      2
b      1
a      1
d      0
dtype: int64
```

Groupby will also show “unused” categories:

```
In [132]: cats = pd.Categorical(["a", "b", "b", "b", "c", "c", "c"], categories=["a", "b", "c", "d"])
In [133]: df = pd.DataFrame({"cats": cats, "values": [1, 2, 2, 2, 3, 4, 5]})
In [134]: df.groupby("cats").mean()
Out[134]:
```

|      | values |
|------|--------|
| cats |        |
| a    | 1.0    |

(continues on next page)

(continued from previous page)

```

b      2.0
c      4.0
d      NaN

In [135]: cats2 = pd.Categorical(["a","a","b","b"], categories=["a","b","c"])

In [136]: df2 = pd.DataFrame({"cats":cats2,"B":["c","d","c","d"], "values":[1,2,3,4]})

In [137]: df2.groupby(["cats","B"]).mean()
Out[137]:
      values
cats B
a     c     1.0
     d     2.0
b     c     3.0
     d     4.0
c     c     NaN
     d     NaN

```

Pivot tables:

```

In [138]: raw_cat = pd.Categorical(["a","a","b","b"], categories=["a","b","c"])

In [139]: df = pd.DataFrame({"A":raw_cat,"B":["c","d","c","d"], "values":[1,2,3,4]})

In [140]: pd.pivot_table(df, values='values', index=['A', 'B'])
Out[140]:
      values
A B
a c         1
  d         2
b c         3
  d         4

```

## 21.8 Data munging

The optimized pandas data access methods `.loc`, `.iloc`, `.at`, and `.iat`, work as normal. The only difference is the return type (for getting) and that only values already in *categories* can be assigned.

### 21.8.1 Getting

If the slicing operation returns either a `DataFrame` or a column of type `Series`, the `category` dtype is preserved.

```

In [141]: idx = pd.Index(["h","i","j","k","l","m","n",])

In [142]: cats = pd.Series(["a","b","b","b","c","c","c"], dtype="category", index=idx)

In [143]: values= [1,2,2,2,3,4,5]

In [144]: df = pd.DataFrame({"cats":cats,"values":values}, index=idx)

In [145]: df.iloc[2:4,:]
Out[145]:

```

(continues on next page)

(continued from previous page)

```

   cats  values
j      b      2
k      b      2

```

```
In [146]: df.iloc[2:4,:].dtypes
```

```

////////////////////////////////////Out[146]:
cats      category
values      int64
dtype: object

```

```
In [147]: df.loc["h":"j", "cats"]
```

```

////////////////////////////////////
↪
h      a
i      b
j      b
Name: cats, dtype: category
Categories (3, object): [a, b, c]

```

```
In [148]: df[df["cats"] == "b"]
```

```

////////////////////////////////////
↪
   cats  values
i      b      2
j      b      2
k      b      2

```

An example where the category type is not preserved is if you take one single row: the resulting `Series` is of dtype `object`:

```
# get the complete "h" row as a Series
```

```
In [149]: df.loc["h", :]
```

```

Out[149]:
cats      a
values      1
Name: h, dtype: object

```

Returning a single item from categorical data will also return the value, not a categorical of length “1”.

```
In [150]: df.iat[0,0]
```

```
Out[150]: 'a'
```

```
In [151]: df["cats"].cat.categories = ["x", "y", "z"]
```

```
In [152]: df.at["h", "cats"] # returns a string
```

```
Out[152]: 'x'
```

---

**Note:** This is in contrast to R’s `factor` function, where `factor(c(1, 2, 3))[1]` returns a single value *factor*.

---

To get a single value `Series` of type `category`, you pass in a list with a single value:

```
In [153]: df.loc[["h"], "cats"]
```

```
Out[153]:
```

```
h      x
```

(continues on next page)

(continued from previous page)

```
Name: cats, dtype: category
Categories (3, object): [x, y, z]
```

## 21.8.2 String and datetime accessors

The accessors `.dt` and `.str` will work if the `s.cat.categories` are of an appropriate type:

```
In [154]: str_s = pd.Series(list('aabb'))

In [155]: str_cat = str_s.astype('category')

In [156]: str_cat
Out[156]:
0    a
1    a
2    b
3    b
dtype: category
Categories (2, object): [a, b]

In [157]: str_cat.str.contains("a")
Out[157]:
0    True
1    True
2   False
3   False
dtype: bool

In [158]: date_s = pd.Series(pd.date_range('1/1/2015', periods=5))

In [159]: date_cat = date_s.astype('category')

In [160]: date_cat
Out[160]:
0    2015-01-01
1    2015-01-02
2    2015-01-03
3    2015-01-04
4    2015-01-05
dtype: category
Categories (5, datetime64[ns]): [2015-01-01, 2015-01-02, 2015-01-03, 2015-01-04, 2015-01-05]

In [161]: date_cat.dt.day
Out[161]:
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

**Note:** The returned Series (or DataFrame) is of the same type as if you used the `.str.<method>` / `.dt.`

<method> on a Series of that type (and not of type *category*!).

---

That means, that the returned values from methods and properties on the accessors of a *Series* and the returned values from methods and properties on the accessors of this *Series* transformed to one of type *category* will be equal:

```
In [162]: ret_s = str_s.str.contains("a")

In [163]: ret_cat = str_cat.str.contains("a")

In [164]: ret_s.dtype == ret_cat.dtype
Out[164]: True

In [165]: ret_s == ret_cat
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[165]:
0      True
1      True
2      True
3      True
dtype: bool
```

---

**Note:** The work is done on the *categories* and then a new *Series* is constructed. This has some performance implication if you have a *Series* of type string, where lots of elements are repeated (i.e. the number of unique elements in the *Series* is a lot smaller than the length of the *Series*). In this case it can be faster to convert the original *Series* to one of type *category* and use `.str.<method>` or `.dt.<property>` on that.

---

### 21.8.3 Setting

Setting values in a categorical column (or *Series*) works as long as the value is included in the *categories*:

```
In [166]: idx = pd.Index(["h", "i", "j", "k", "l", "m", "n"])

In [167]: cats = pd.Categorical(["a", "a", "a", "a", "a", "a", "a"], categories=["a", "b"])

In [168]: values = [1, 1, 1, 1, 1, 1, 1]

In [169]: df = pd.DataFrame({"cats": cats, "values": values}, index=idx)

In [170]: df.iloc[2:4, :] = [["b", 2], ["b", 2]]

In [171]: df
Out[171]:
   cats  values
h     a        1
i     a        1
j     b        2
k     b        2
l     a        1
m     a        1
n     a        1

In [172]: try:
.....:     df.iloc[2:4, :] = [["c", 3], ["c", 3]]
```

(continues on next page)

(continued from previous page)

```

.....: except ValueError as e:
.....:     print("ValueError: " + str(e))
.....:
////////////////////////////////////
↪Cannot setitem on a Categorical with a new category, set the categories first

```

Setting values by assigning categorical data will also check that the *categories* match:

```

In [173]: df.loc["j":"k", "cats"] = pd.Categorical(["a", "a"], categories=["a", "b"])

In [174]: df
Out[174]:
   cats  values
h     a        1
i     a        1
j     a        2
k     a        2
l     a        1
m     a        1
n     a        1

In [175]: try:
.....:     df.loc["j":"k", "cats"] = pd.Categorical(["b", "b"], categories=["a", "b",
↪ "c"])
.....: except ValueError as e:
.....:     print("ValueError: " + str(e))
.....:
////////////////////////////////////
↪Cannot set a Categorical with another, without identical categories

```

Assigning a Categorical to parts of a column of other types will use the values:

```

In [176]: df = pd.DataFrame({"a": [1, 1, 1, 1, 1], "b": ["a", "a", "a", "a", "a"]})

In [177]: df.loc[1:2, "a"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [178]: df.loc[2:3, "b"] = pd.Categorical(["b", "b"], categories=["a", "b"])

In [179]: df
Out[179]:
   a  b
0  1  a
1  b  a
2  b  b
3  1  b
4  1  a

In [180]: df.dtypes
Out[180]:
a    object
b    object
dtype: object

```

## 21.8.4 Merging

You can concat two DataFrames containing categorical data together, but the categories of these categoricals need to be the same:

```
In [181]: cat = pd.Series(["a", "b"], dtype="category")

In [182]: vals = [1, 2]

In [183]: df = pd.DataFrame({"cats": cat, "vals": vals})

In [184]: res = pd.concat([df, df])

In [185]: res
Out[185]:
   cats  vals
0     a     1
1     b     2
0     a     1
1     b     2

In [186]: res.dtypes
Out[186]:
cats      category
vals      int64
dtype: object
```

In this case the categories are not the same, and therefore an error is raised:

```
In [187]: df_different = df.copy()

In [188]: df_different["cats"].cat.categories = ["c", "d"]

In [189]: try:
.....:     pd.concat([df, df_different])
.....: except ValueError as e:
.....:     print("ValueError: " + str(e))
.....:
```

The same applies to `df.append(df_different)`.

See also the section on *merge dtypes* for notes about preserving merge dtypes and performance.

## 21.8.5 Unioning

New in version 0.19.0.

If you want to combine categoricals that do not necessarily have the same categories, the `union_categoricals()` function will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```
In [190]: from pandas.api.types import union_categoricals

In [191]: a = pd.Categorical(["b", "c"])

In [192]: b = pd.Categorical(["a", "b"])

In [193]: union_categoricals([a, b])
```

(continues on next page)



(continued from previous page)

```
Out [193]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

By default, the resulting categories will be ordered as they appear in the data. If you want the categories to be lexicographically sorted, use `sort_categories=True` argument.

```
In [194]: union_categoricals([a, b], sort_categories=True)
Out [194]:
[b, c, a, b]
Categories (3, object): [a, b, c]
```

`union_categoricals` also works with the “easy” case of combining two categoricals of the same categories and order information (e.g. what you could also append for).

```
In [195]: a = pd.Categorical(["a", "b"], ordered=True)
In [196]: b = pd.Categorical(["a", "b", "a"], ordered=True)
In [197]: union_categoricals([a, b])
Out [197]:
[a, b, a, b, a]
Categories (2, object): [a < b]
```

The below raises `TypeError` because the categories are ordered and not identical.

```
In [1]: a = pd.Categorical(["a", "b"], ordered=True)
In [2]: b = pd.Categorical(["a", "b", "c"], ordered=True)
In [3]: union_categoricals([a, b])
Out [3]:
TypeError: to union ordered Categoricals, all categories must be the same
```

New in version 0.20.0.

Ordered categoricals with different categories or orderings can be combined by using the `ignore_ordered=True` argument.

```
In [198]: a = pd.Categorical(["a", "b", "c"], ordered=True)
In [199]: b = pd.Categorical(["c", "b", "a"], ordered=True)
In [200]: union_categoricals([a, b], ignore_order=True)
Out [200]:
[a, b, c, c, b, a]
Categories (3, object): [a, b, c]
```

`union_categoricals()` also works with a `CategoricalIndex`, or `Series` containing categorical data, but note that the resulting array will always be a plain `Categorical`:

```
In [201]: a = pd.Series(["b", "c"], dtype='category')
In [202]: b = pd.Series(["a", "b"], dtype='category')
In [203]: union_categoricals([a, b])
Out [203]:
[b, c, a, b]
Categories (3, object): [b, c, a]
```

**Note:** `union_categoricals` may recode the integer codes for categories when combining categoricals. This is likely what you want, but if you are relying on the exact numbering of the categories, be aware.

```
In [204]: c1 = pd.Categorical(["b", "c"])

In [205]: c2 = pd.Categorical(["a", "b"])

In [206]: c1
Out[206]:
[b, c]
Categories (2, object): [b, c]

# "b" is coded to 0
In [207]: c1.codes
Out[207]: array([0, 1], dtype=int8)

In [208]: c2
Out[208]:
[a, b]
Categories (2, object): [a, b]

# "b" is coded to 1
In [209]: c2.codes
Out[209]: array([0, 1], dtype=int8)

In [210]: c = union_categoricals([c1, c2])

In [211]: c
Out[211]:
[b, c, a, b]
Categories (3, object): [b, c, a]

# "b" is coded to 0 throughout, same as c1, different from c2
In [212]: c.codes
Out[212]: array([0, 1, 2, 0], dtype=int8)
```

## 21.8.6 Concatenation

This section describes concatenations specific to `category` dtype. See *Concatenating objects* for general description.

By default, `Series` or `DataFrame` concatenation which contains the same categories results in `category` dtype, otherwise results in `object` dtype. Use `.astype` or `union_categoricals` to get category result.

```
# same categories
In [213]: s1 = pd.Series(['a', 'b'], dtype='category')

In [214]: s2 = pd.Series(['a', 'b', 'a'], dtype='category')

In [215]: pd.concat([s1, s2])
Out[215]:
0      a
```

(continues on next page)

(continued from previous page)

```

1      b
0      a
1      b
2      a
dtype: category
Categories (2, object): [a, b]

# different categories
In [216]: s3 = pd.Series(['b', 'c'], dtype='category')

In [217]: pd.concat([s1, s3])
Out[217]:
0      a
1      b
0      b
1      c
dtype: object

In [218]: pd.concat([s1, s3]).astype('category')
\\Out[218]:
0      a
1      b
0      b
1      c
dtype: category
Categories (3, object): [a, b, c]

In [219]: union_categoricals([s1.values, s3.values])
\\
↪
[a, b, b, c]
Categories (3, object): [a, b, c]

```

Following table summarizes the results of `Categoricals` related concatenations.

| arg1     | arg2   | result                     |
|----------|--|----------------------------|
| category | category (identical categories)                        | category                   |
| category | category (different categories, both not ordered)      | object (dtype is inferred) |
| category | category (different categories, either one is ordered) | object (dtype is inferred) |
| category | not category   | object (dtype is inferred) |

## 21.9 Getting Data In/Out

You can write data that contains `category` dtypes to a `HDFStore`. See [here](#) for an example and caveats.

It is also possible to write data to and reading data from *Stata* format files. See [here](#) for an example and caveats.

Writing to a CSV file will convert the data, effectively removing any information about the categorical (categories and ordering). So if you read back the CSV file you have to convert the relevant columns back to `category` and assign the right categories and categories ordering.

```

In [220]: s = pd.Series(pd.Categorical(['a', 'b', 'b', 'a', 'a', 'd']))

# rename the categories

```

(continues on next page)

(continued from previous page)

```

In [221]: s.cat.categories = ["very good", "good", "bad"]

# reorder the categories and add missing categories
In [222]: s = s.cat.set_categories(["very bad", "bad", "medium", "good", "very good"])

In [223]: df = pd.DataFrame({"cats":s, "vals":[1,2,3,4,5,6]})

In [224]: csv = StringIO()

In [225]: df.to_csv(csv)

In [226]: df2 = pd.read_csv(StringIO(csv.getvalue()))

In [227]: df2.dtypes
Out[227]:
Unnamed: 0      int64
cats           object
vals           int64
dtype: object

In [228]: df2["cats"]
Out[228]:
↪
0      very good
1         good
2         good
3      very good
4      very good
5         bad
Name: cats, dtype: object

# Redo the category
In [229]: df2["cats"] = df2["cats"].astype("category")

In [230]: df2["cats"].cat.set_categories(["very bad", "bad", "medium", "good", "very
↪good"],
.....:                                     inplace=True)
.....:

In [231]: df2.dtypes
Out[231]:
Unnamed: 0      int64
cats           category
vals           int64
dtype: object

In [232]: df2["cats"]
Out[232]:
↪
0      very good
1         good
2         good
3      very good
4      very good
5         bad
Name: cats, dtype: category
Categories (5, object): [very bad, bad, medium, good, very good]

```

The same holds for writing to a SQL database with `to_sql`.

## 21.10 Missing Data

pandas primarily uses the value `np.nan` to represent missing data. It is by default not included in computations. See the [Missing Data section](#).

Missing values should **not** be included in the Categorical's `categories`, only in the `values`. Instead, it is understood that NaN is different, and is always a possibility. When working with the Categorical's `codes`, missing values will always have a code of `-1`.

```
In [233]: s = pd.Series(["a", "b", np.nan, "a"], dtype="category")

# only two categories
In [234]: s
Out[234]:
0      a
1      b
2     NaN
3      a
dtype: category
Categories (2, object): [a, b]

In [235]: s.cat.codes
Out[235]:
0      0
1      1
2     -1
3      0
dtype: int8
```

Methods for working with missing data, e.g. `isna()`, `fillna()`, `dropna()`, all work normally:

```
In [236]: s = pd.Series(["a", "b", np.nan], dtype="category")

In [237]: s
Out[237]:
0      a
1      b
2     NaN
dtype: category
Categories (2, object): [a, b]

In [238]: pd.isna(s)
Out[238]:
0    False
1    False
2     True
dtype: bool

In [239]: s.fillna("a")
Out[239]:
0      a
```

(continues on next page)

(continued from previous page)

```
1      b
2      a
dtype: category
Categories (2, object): [a, b]
```

## 21.11 Differences to R's *factor*

The following differences to R's factor functions can be observed:

- R's *levels* are named *categories*.
- R's *levels* are always of type string, while *categories* in pandas can be of any dtype.
- It's not possible to specify labels at creation time. Use `s.cat.rename_categories(new_labels)` afterwards.
- In contrast to R's *factor* function, using categorical data as the sole input to create a new categorical series will *not* remove unused categories but create a new categorical series which is equal to the passed in one!
- R allows for missing values to be included in its *levels* (pandas' *categories*). Pandas does not allow *NaN* categories, but missing values can still be in the *values*.

## 21.12 Gotchas

### 21.12.1 Memory Usage

The memory usage of a `Categorical` is proportional to the number of categories plus the length of the data. In contrast, an `object` dtype is a constant times the length of the data.

```
In [240]: s = pd.Series(['foo', 'bar']*1000)

# object dtype
In [241]: s.nbytes
Out[241]: 16000

# category dtype
In [242]: s.astype('category').nbytes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[242]: 2016
```

---

**Note:** If the number of categories approaches the length of the data, the `Categorical` will use nearly the same or more memory than an equivalent `object` dtype representation.

```
In [243]: s = pd.Series(['foo%04d' % i for i in range(2000)])

# object dtype
In [244]: s.nbytes
Out[244]: 16000

# category dtype
In [245]: s.astype('category').nbytes
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[245]: 20000
```

### 21.12.2 *Categorical* is not a *numpy* array

Currently, categorical data and the underlying `Categorical` is implemented as a Python object and not as a low-level NumPy array dtype. This leads to some problems.

NumPy itself doesn't know about the new *dtype*:

```
In [246]: try:
.....:     np.dtype("category")
.....: except TypeError as e:
.....:     print("TypeError: " + str(e))
.....:
TypeError: data type "category" not understood

In [247]: dtype = pd.Categorical(["a"]).dtype

In [248]: try:
.....:     np.dtype(dtype)
.....: except TypeError as e:
.....:     print("TypeError: " + str(e))
.....:
TypeError: data type not understood
```

Dtype comparisons work:

```
In [249]: dtype == np.str_
Out[249]: False

In [250]: np.str_ == dtype
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[250]: False
```

To check if a Series contains Categorical data, use `hasattr(s, 'cat')`:

```
In [251]: hasattr(pd.Series(['a'], dtype='category'), 'cat')
Out[251]: True

In [252]: hasattr(pd.Series(['a']), 'cat')
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[252]: False
```

Using NumPy functions on a Series of type category should not work as *Categoricals* are not numeric data (even in the case that `.categories` is numeric).

```
In [253]: s = pd.Series(pd.Categorical([1,2,3,4]))

In [254]: try:
.....:     np.sum(s)
.....: except TypeError as e:
.....:     print("TypeError: " + str(e))
.....:
TypeError: Categorical cannot perform the operation sum
```

---

**Note:** If such a function works, please file a bug at <https://github.com/pandas-dev/pandas!>

---

### 21.12.3 dtype in apply

Pandas currently does not preserve the dtype in apply functions: If you apply along rows you get a *Series* of *object* dtype (same as getting a row -> getting one element will return a basic type) and applying along columns will also convert to object.

```
In [255]: df = pd.DataFrame({"a": [1, 2, 3, 4],
.....:                       "b": ["a", "b", "c", "d"],
.....:                       "cats": pd.Categorical([1, 2, 3, 2])})
.....:
```

```
In [256]: df.apply(lambda row: type(row["cats"]), axis=1)
```

```
Out [256]:
0    <class 'int'>
1    <class 'int'>
2    <class 'int'>
3    <class 'int'>
dtype: object
```

```
In [257]: df.apply(lambda col: col.dtype, axis=0)
```

### 21.12.4 Categorical Index

`CategoricalIndex` is a type of index that is useful for supporting indexing with duplicates. This is a container around a `Categorical` and allows efficient indexing and storage of an index with a large number of duplicated elements. See the [advanced indexing docs](#) for a more detailed explanation.

Setting the index will create a `CategoricalIndex`:

```
In [258]: cats = pd.Categorical([1,2,3,4], categories=[4,2,3,1])
```

```
In [259]: strings = ["a", "b", "c", "d"]
```

```
In [260]: values = [4, 2, 3, 1]
```

```
In [261]: df = pd.DataFrame({"strings":strings, "values":values}, index=cats)
```

```
In [262]: df.index
```

```
Out [262]: CategoricalIndex([1, 2, 3, 4], categories=[4, 2, 3, 1], ordered=False,
dtype='category')
```

```
# This now sorts by the categories order
```

```
In [263]: df.sort_index()
```

```

\\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\
↪
strings  values
4        d        1
2        b        2
3        c        3
1        a        4

```



### 21.12.5 Side Effects

Constructing a Series from a Categorical will not copy the input Categorical. This means that changes to the Series will in most cases change the original Categorical:

```
In [264]: cat = pd.Categorical([1,2,3,10], categories=[1,2,3,4,10])

In [265]: s = pd.Series(cat, name="cat")

In [266]: cat
Out[266]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [267]: s.iloc[0:2] = 10

In [268]: cat
Out[268]:
[10, 10, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [269]: df = pd.DataFrame(s)

In [270]: df["cat"].cat.categories = [1,2,3,4,5]

In [271]: cat
Out[271]:
[5, 5, 3, 5]
Categories (5, int64): [1, 2, 3, 4, 5]
```

Use `copy=True` to prevent such a behaviour or simply don't reuse Categoricals:

```
In [272]: cat = pd.Categorical([1,2,3,10], categories=[1,2,3,4,10])

In [273]: s = pd.Series(cat, name="cat", copy=True)

In [274]: cat
Out[274]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]

In [275]: s.iloc[0:2] = 10

In [276]: cat
Out[276]:
[1, 2, 3, 10]
Categories (5, int64): [1, 2, 3, 4, 10]
```

**Note:** This also happens in some cases when you supply a NumPy array instead of a Categorical: using an int array (e.g. `np.array([1,2,3,4])`) will exhibit the same behavior, while using a string array (e.g. `np.array(["a","b","c","a"])`) will not.



## VISUALIZATION

We use the standard convention for referencing the matplotlib API:

```
In [1]: import matplotlib.pyplot as plt
```

We provide the basics in pandas to easily create decent looking plots. See the *ecosystem* section for visualization libraries that go beyond the basics documented here.

---

**Note:** All calls to `np.random` are seeded with 123456.

---

### 22.1 Basic Plotting: `plot`

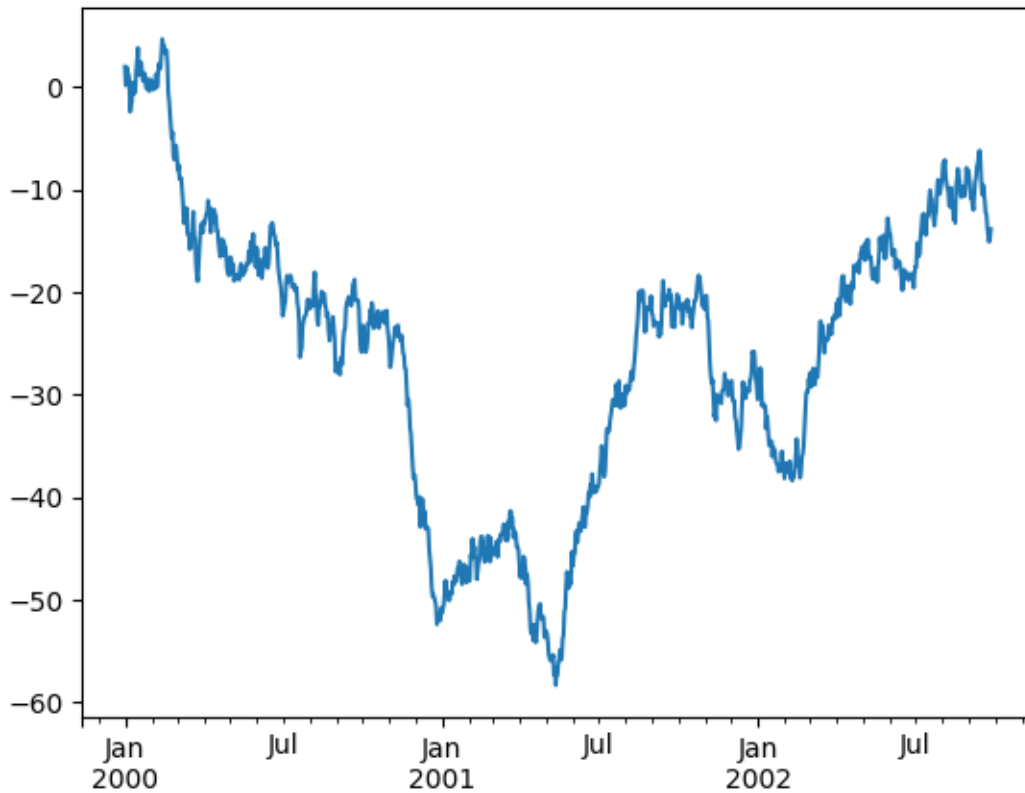
We will demonstrate the basics, see the *cookbook* for some advanced strategies.

The `plot` method on `Series` and `DataFrame` is just a simple wrapper around `plt.plot()`:

```
In [2]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
↳ periods=1000))

In [3]: ts = ts.cumsum()

In [4]: ts.plot()
Out[4]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2e3b0ba8>
```



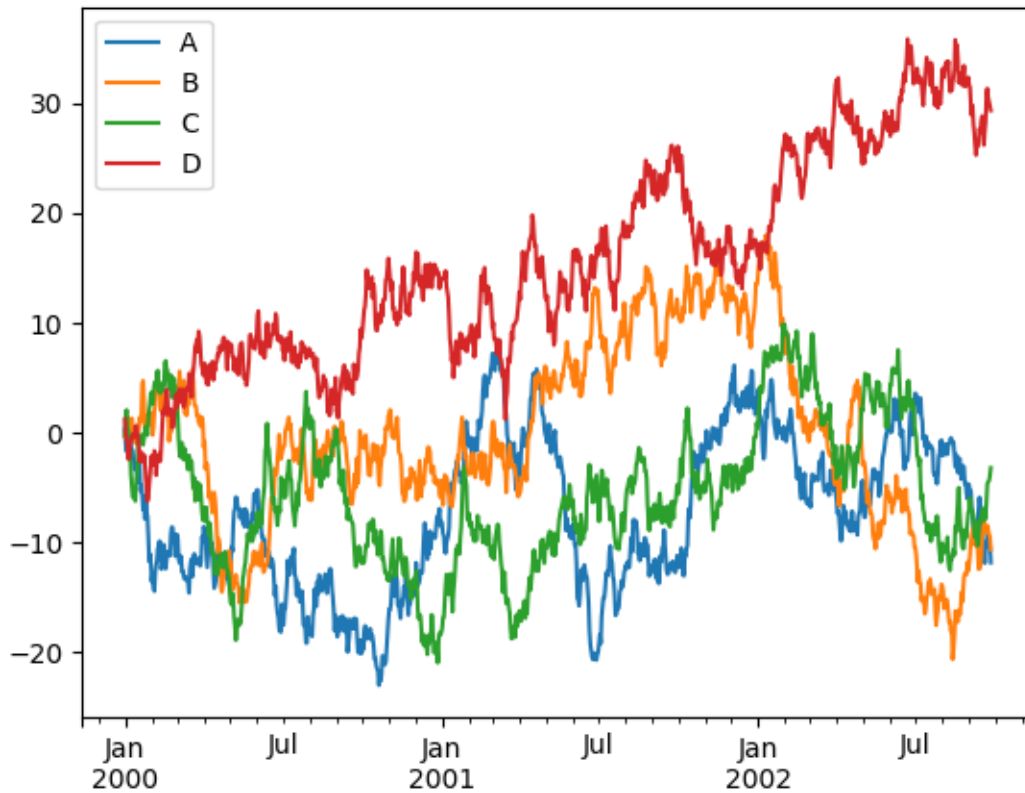
If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above.

On `DataFrame`, `plot()` is a convenience to plot all of the columns with labels:

```
In [5]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list('ABCD
↳'))

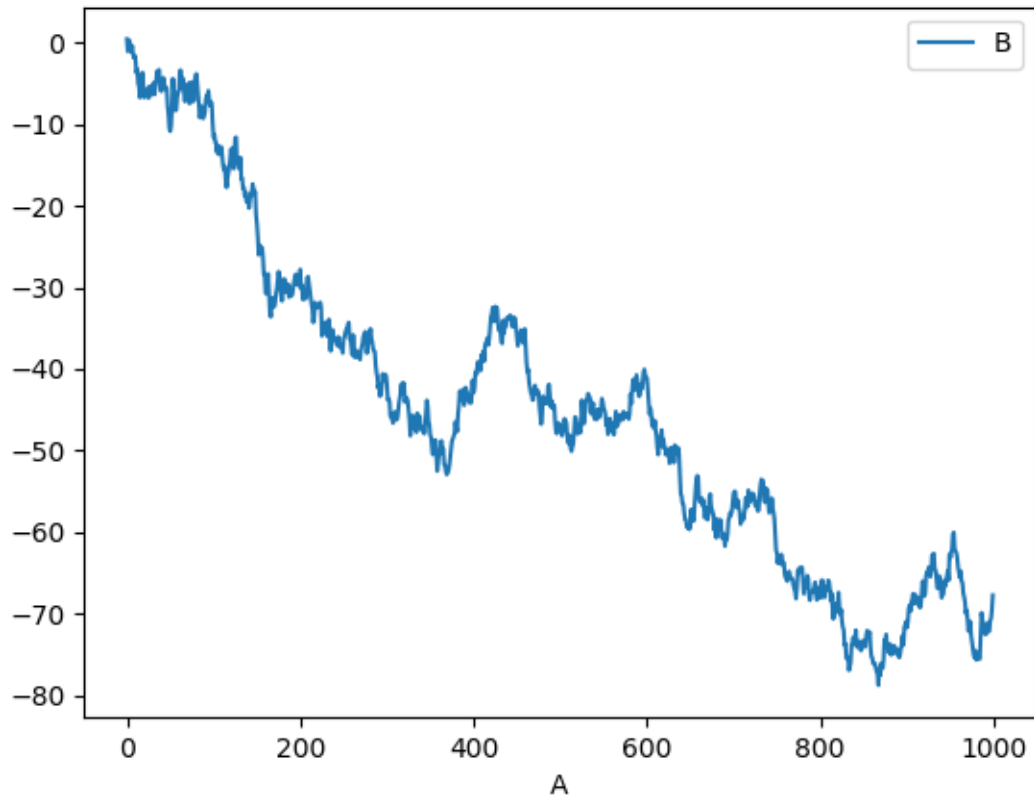
In [6]: df = df.cumsum()

In [7]: plt.figure(); df.plot();
```



You can plot one column versus another using the *x* and *y* keywords in `plot()`:

```
In [8]: df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()
In [9]: df3['A'] = pd.Series(list(range(len(df))))
In [10]: df3.plot(x='A', y='B')
Out[10]: <matplotlib.axes._subplots.AxesSubplot at 0x1c38114e10>
```



---

**Note:** For more formatting and styling options, see *formatting* below.

---

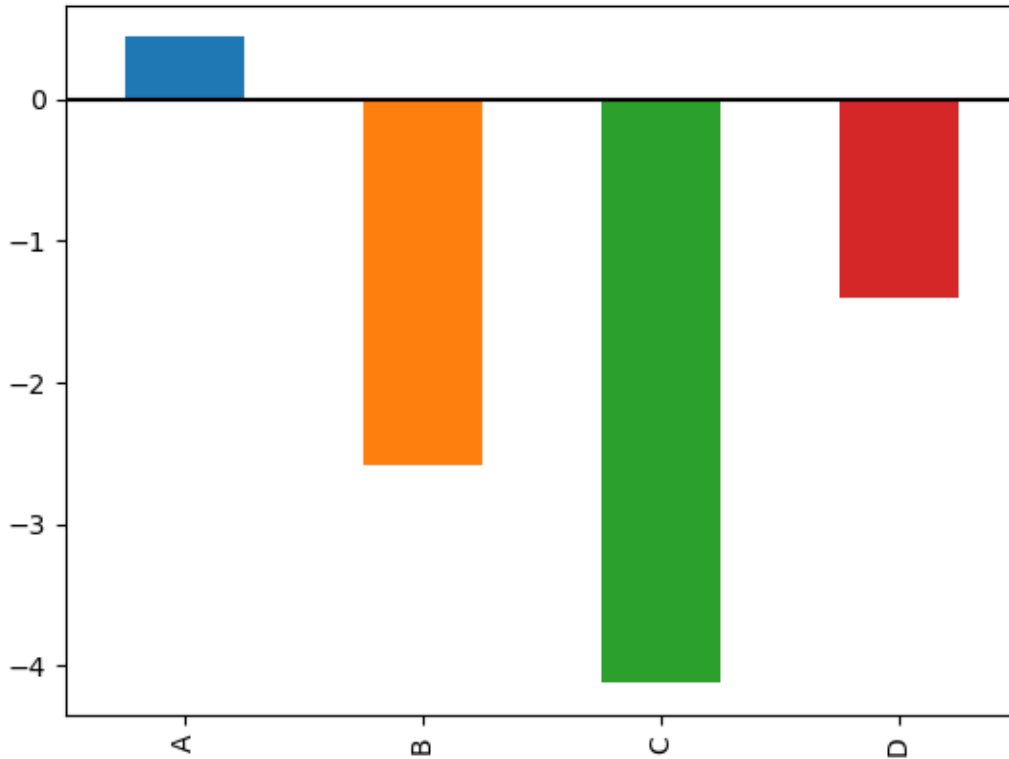
## 22.2 Other Plots

Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the `kind` keyword argument to `plot()`, and include:

- `'bar'` or `'barh'` for bar plots
- `'hist'` for histogram
- `'box'` for boxplot
- `'kde'` or `'density'` for density plots
- `'area'` for area plots
- `'scatter'` for scatter plots
- `'hexbin'` for hexagonal bin plots
- `'pie'` for pie plots

For example, a bar plot can be created the following way:

```
In [11]: plt.figure();
In [12]: df.iloc[5].plot(kind='bar');
```



You can also create these other plots using the methods `DataFrame.plot.<kind>` instead of providing the `kind` keyword argument. This makes it easier to discover plot methods and the specific arguments they use:

```
In [13]: df = pd.DataFrame()

In [14]: df.plot.<TAB>
df.plot.area      df.plot.barh      df.plot.density  df.plot.hist      df.plot.line
↳ df.plot.scatter
df.plot.bar        df.plot.box        df.plot.hexbin   df.plot.kde        df.plot.pie
```

In addition to these kinds, there are the `DataFrame.hist()`, and `DataFrame.boxplot()` methods, which use a separate interface.

Finally, there are several *plotting functions* in `pandas.plotting` that take a *Series* or *DataFrame* as an argument. These include:

- *Scatter Matrix*
- *Andrews Curves*
- *Parallel Coordinates*
- *Lag Plot*

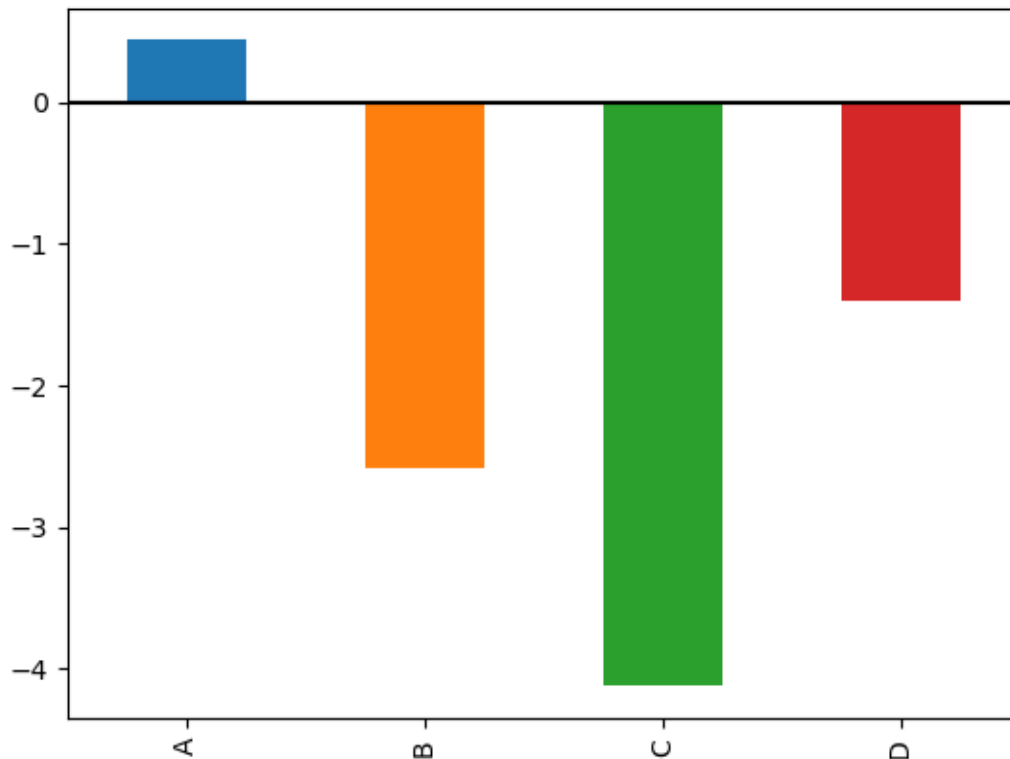
- *Autocorrelation Plot*
- *Bootstrap Plot*
- *RadViz*

Plots may also be adorned with *errorbars* or *tables*.

## 22.2.1 Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

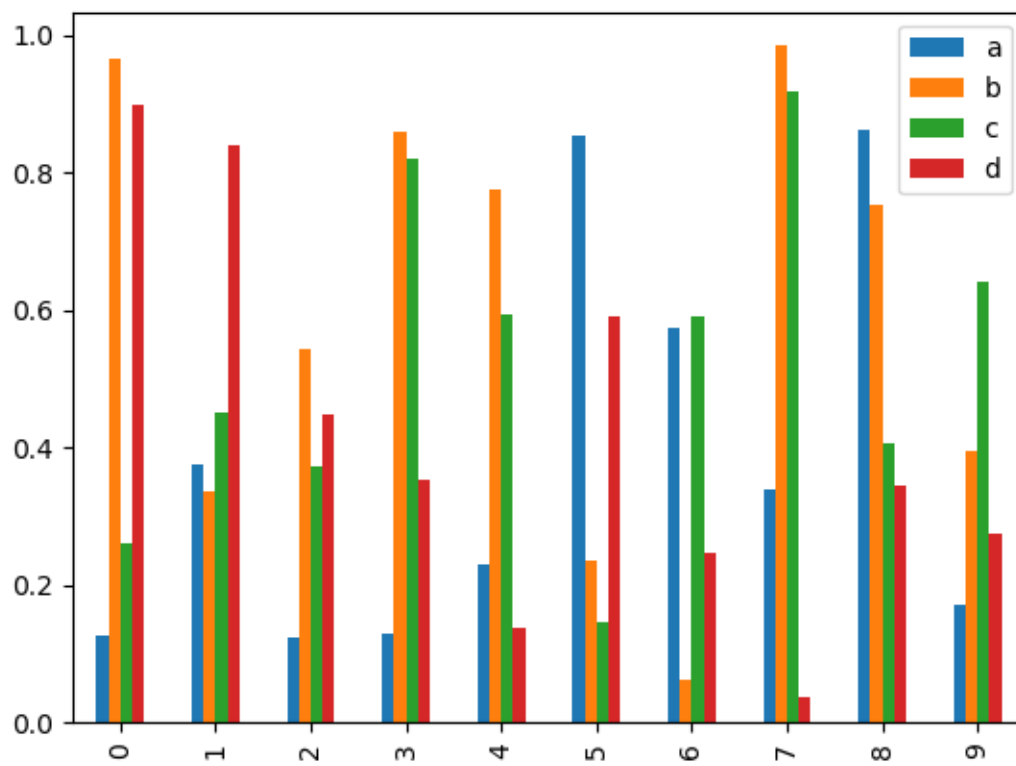
```
In [15]: plt.figure();  
In [16]: df.iloc[5].plot.bar(); plt.axhline(0, color='k')  
Out[16]: <matplotlib.lines.Line2D at 0x1c33f082e8>
```



Calling a DataFrame's `plot.bar()` method produces a multiple bar plot:

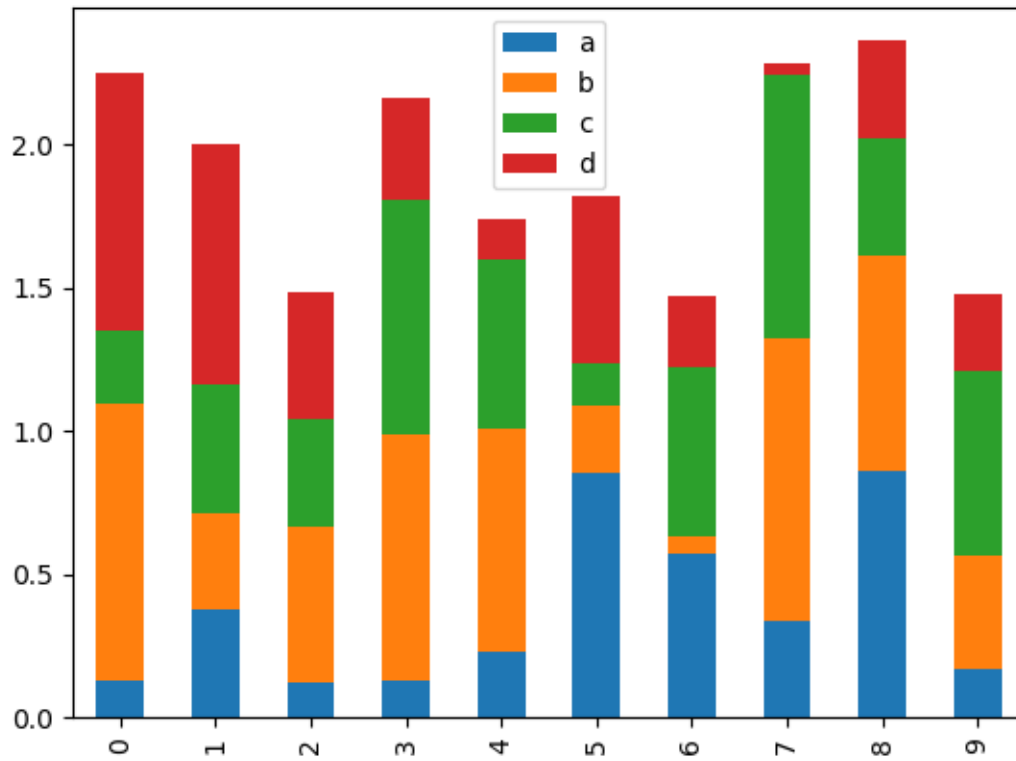
```
In [17]: df2 = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])  
In [18]: df2.plot.bar();
```





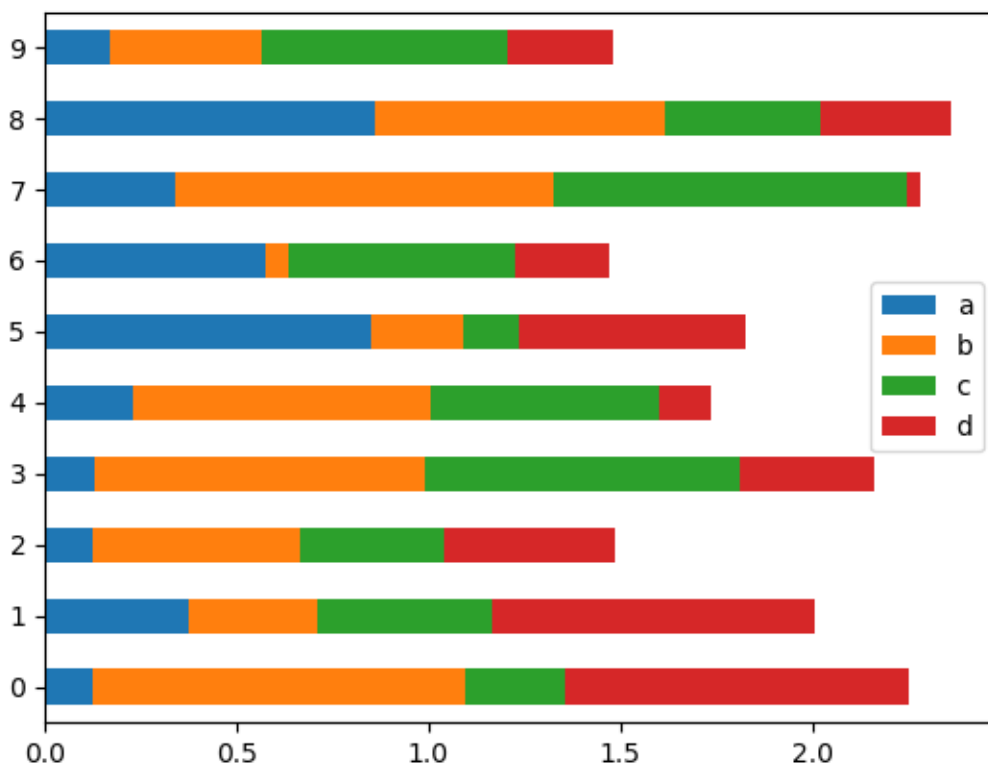
To produce a stacked bar plot, pass `stacked=True`:

```
In [19]: df2.plot.bar(stacked=True);
```



To get horizontal bar plots, use the `barh` method:

```
In [20]: df2.plot.barh(stacked=True);
```



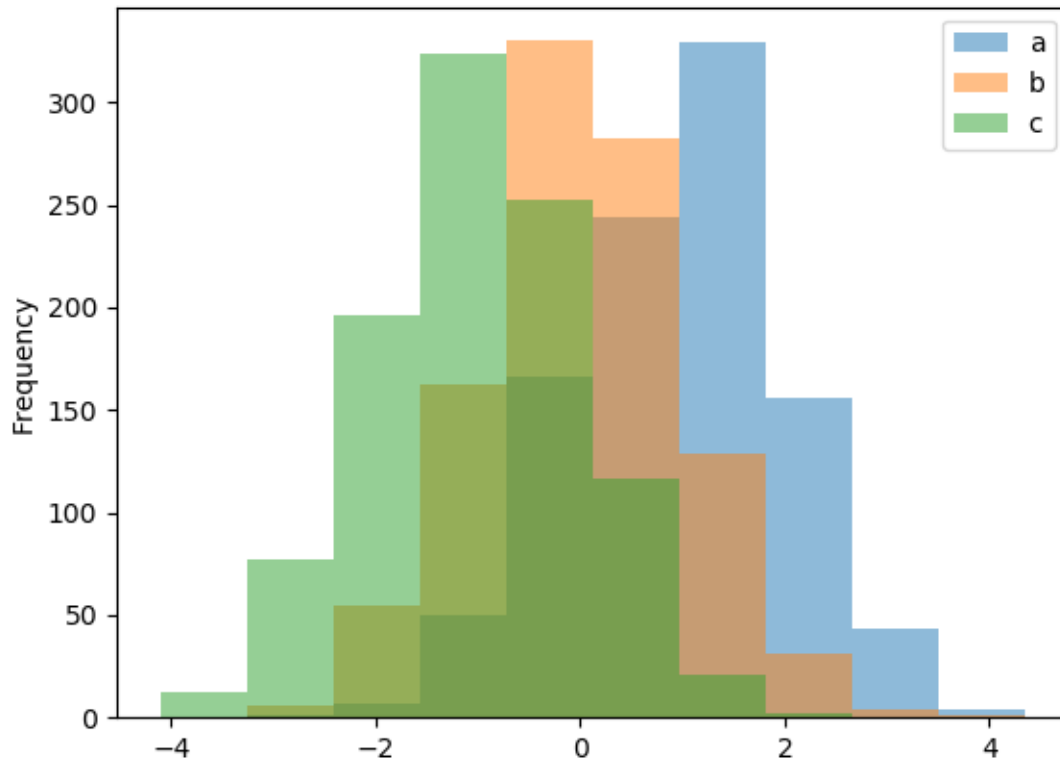
### 22.2.2 Histograms

Histograms can be drawn by using the `DataFrame.plot.hist()` and `Series.plot.hist()` methods.

```
In [21]: df4 = pd.DataFrame({'a': np.random.randn(1000) + 1, 'b': np.random.
↳randn(1000),
      ....:                  'c': np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])
      ....:

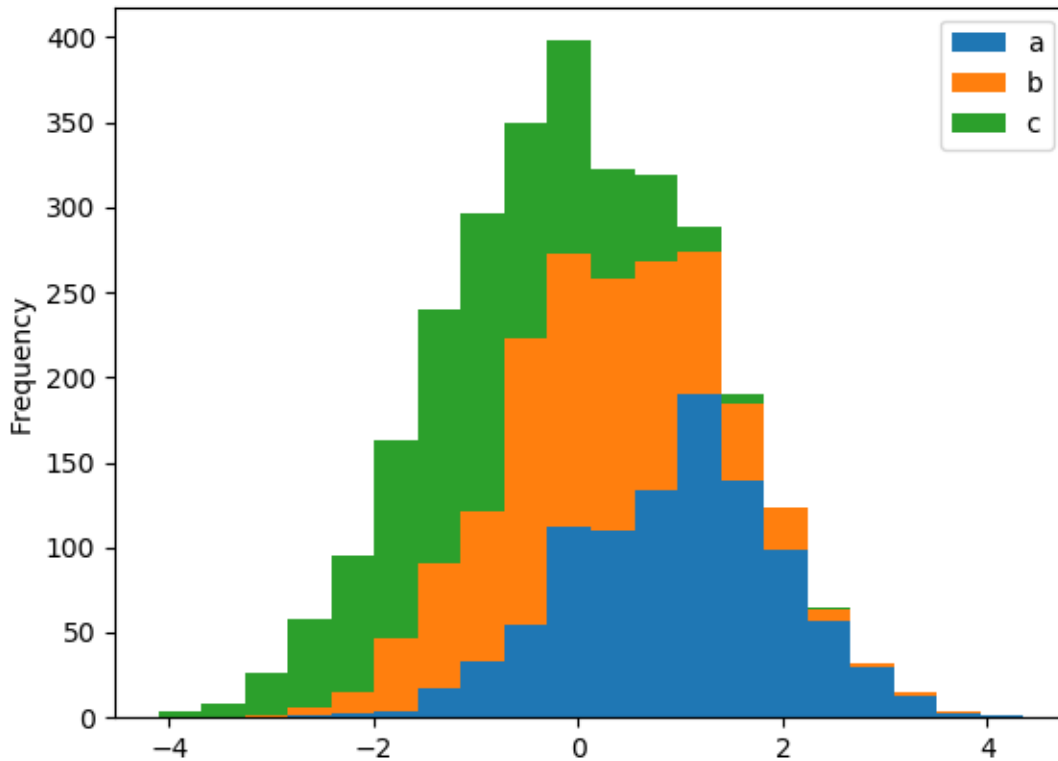
In [22]: plt.figure();

In [23]: df4.plot.hist(alpha=0.5)
Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0x1c36abc470>
```



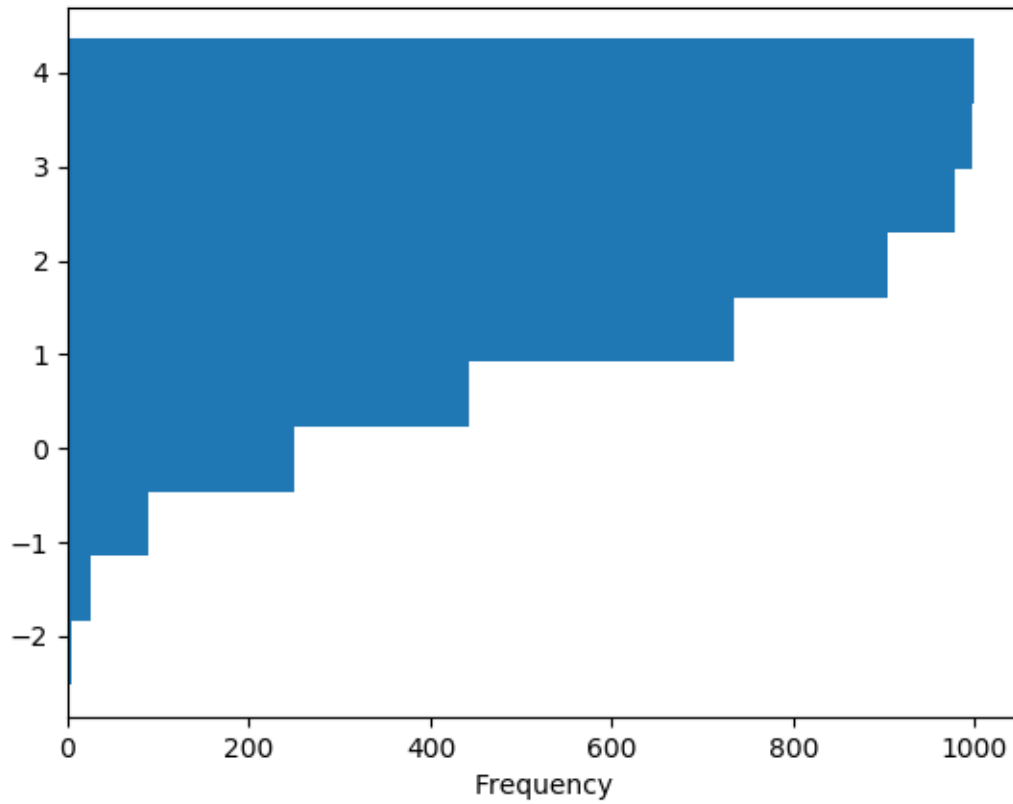
A histogram can be stacked using `stacked=True`. Bin size can be changed using the `bins` keyword.

```
In [24]: plt.figure();  
  
In [25]: df4.plot.hist(stacked=True, bins=20)  
Out[25]: <matplotlib.axes._subplots.AxesSubplot at 0x1c36ac4940>
```



You can pass other keywords supported by matplotlib `hist`. For example, horizontal and cumulative histograms can be drawn by `orientation='horizontal'` and `cumulative=True`.

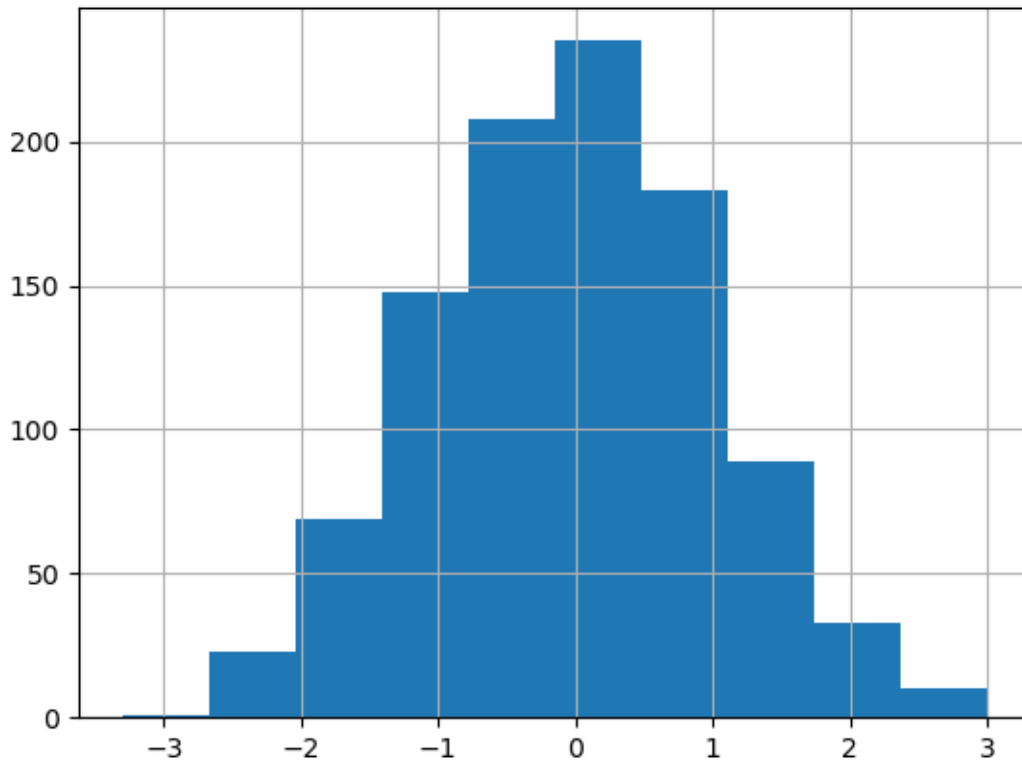
```
In [26]: plt.figure();  
  
In [27]: df4['a'].plot.hist(orientation='horizontal', cumulative=True)  
Out[27]: <matplotlib.axes._subplots.AxesSubplot at 0x1c369362b0>
```



See the [hist](#) method and the [matplotlib hist documentation](#) for more.

The existing interface `DataFrame.hist` to plot histogram still can be used.

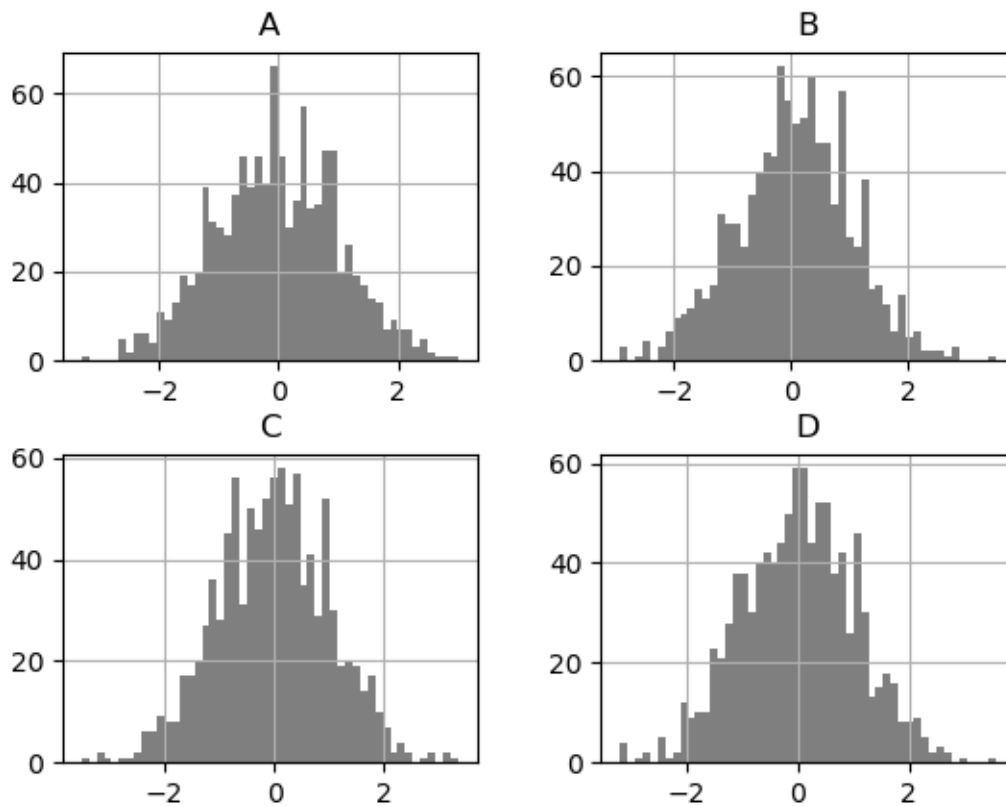
```
In [28]: plt.figure();  
  
In [29]: df['A'].diff().hist()  
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1c388ceeb8>
```



`DataFrame.hist()` plots the histograms of the columns on multiple subplots:

```
In [30]: plt.figure()
Out[30]: <Figure size 640x480 with 0 Axes>

In [31]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[31]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c2a6eda20>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c37637860>],
       [<matplotlib.axes._subplots.AxesSubplot object at 0x1c3764ab70>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c37d4be80>]],
      dtype=object)
```

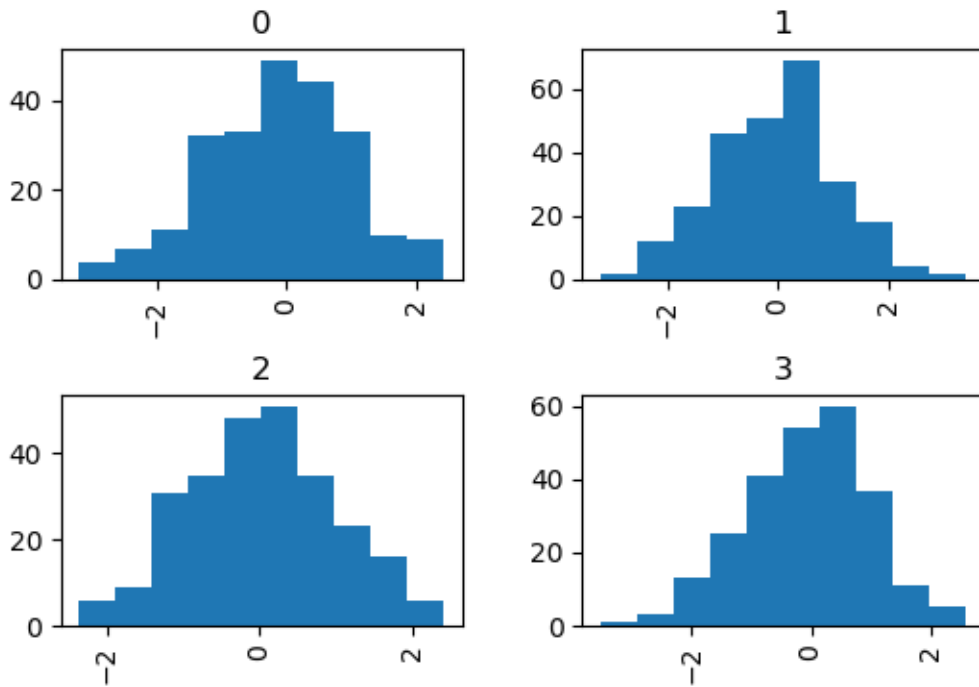


The `by` keyword can be specified to plot grouped histograms:

```
In [32]: data = pd.Series(np.random.randn(1000))

In [33]: data.hist(by=np.random.randint(0, 4, 1000), figsize=(6, 4))
Out[33]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c36be2d30>,
      <matplotlib.axes._subplots.AxesSubplot object at 0x1c2efde940>],
      [<matplotlib.axes._subplots.AxesSubplot object at 0x1c2f002e48>,
      <matplotlib.axes._subplots.AxesSubplot object at 0x1c33d2c198>]],
      dtype=object)
```



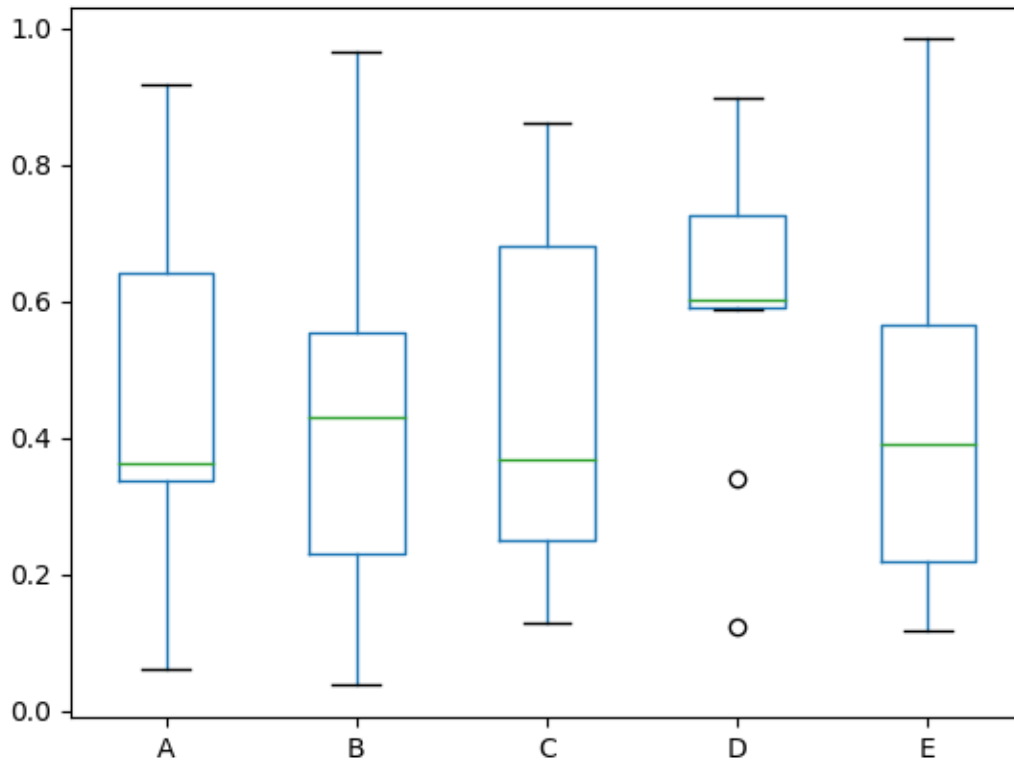


### 22.2.3 Box Plots

Boxplot can be drawn calling `Series.plot.box()` and `DataFrame.plot.box()`, or `DataFrame.boxplot()` to visualize the distribution of values within each column.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on  $[0,1)$ .

```
In [34]: df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C', 'D', 'E'])
In [35]: df.plot.box()
Out[35]: <matplotlib.axes._subplots.AxesSubplot at 0x1c38838f98>
```



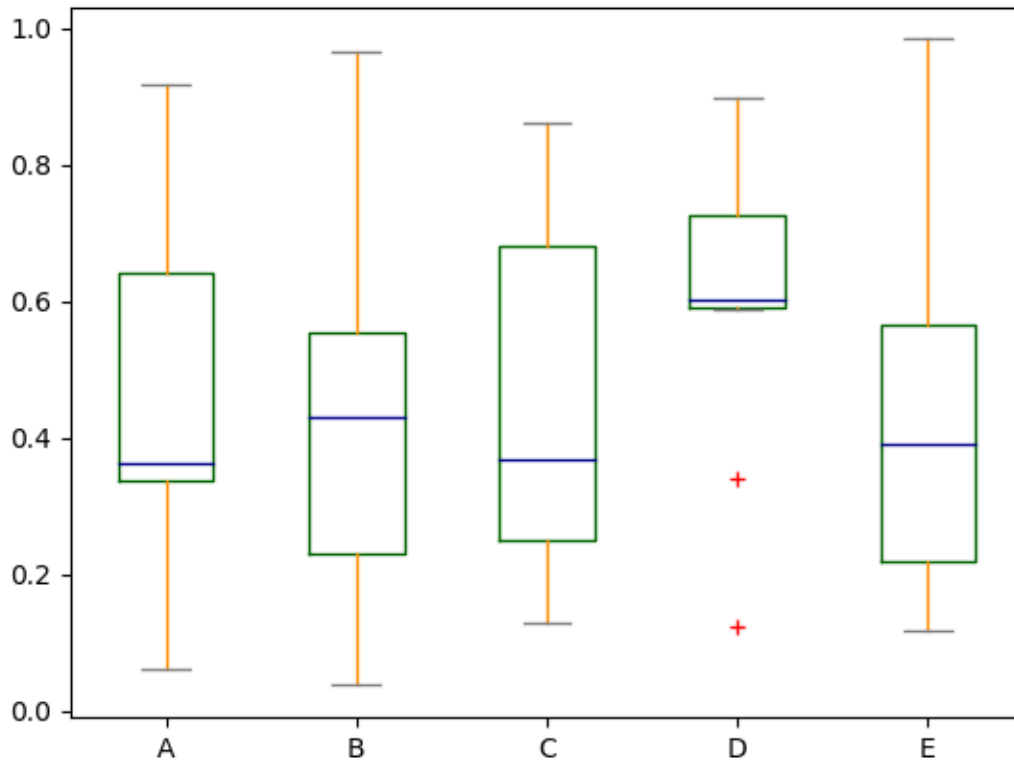
Boxplot can be colored by passing `color` keyword. You can pass a dict whose keys are `boxes`, `whiskers`, `medians` and `caps`. If some keys are missing in the dict, default colors are used for the corresponding artists. Also, boxplot has `sym` keyword to specify fliers style.

When you pass other type of arguments via `color` keyword, it will be directly passed to matplotlib for all the `boxes`, `whiskers`, `medians` and `caps` colorization.

The colors are applied to every boxes to be drawn. If you want more complicated colorization, you can get each drawn artists by passing `return_type`.

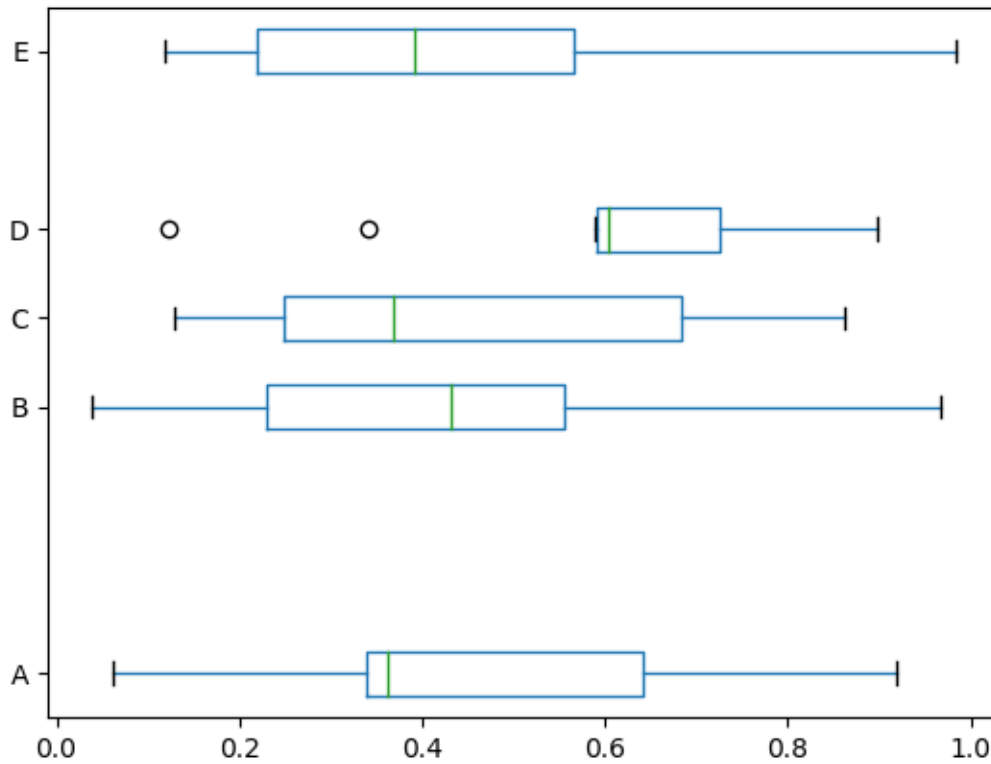
```
In [36]: color = dict(boxes='DarkGreen', whiskers='DarkOrange',
.....:               medians='DarkBlue', caps='Gray')
.....:

In [37]: df.plot.box(color=color, sym='r+')
Out [37]: <matplotlib.axes._subplots.AxesSubplot at 0x1c378c4198>
```



Also, you can pass other keywords supported by matplotlib `boxplot`. For example, horizontal and custom-positioned boxplot can be drawn by `vert=False` and `positions` keywords.

```
In [38]: df.plot.box(vert=False, positions=[1, 4, 5, 6, 8])
Out [38]: <matplotlib.axes._subplots.AxesSubplot at 0x1c36a6ae10>
```



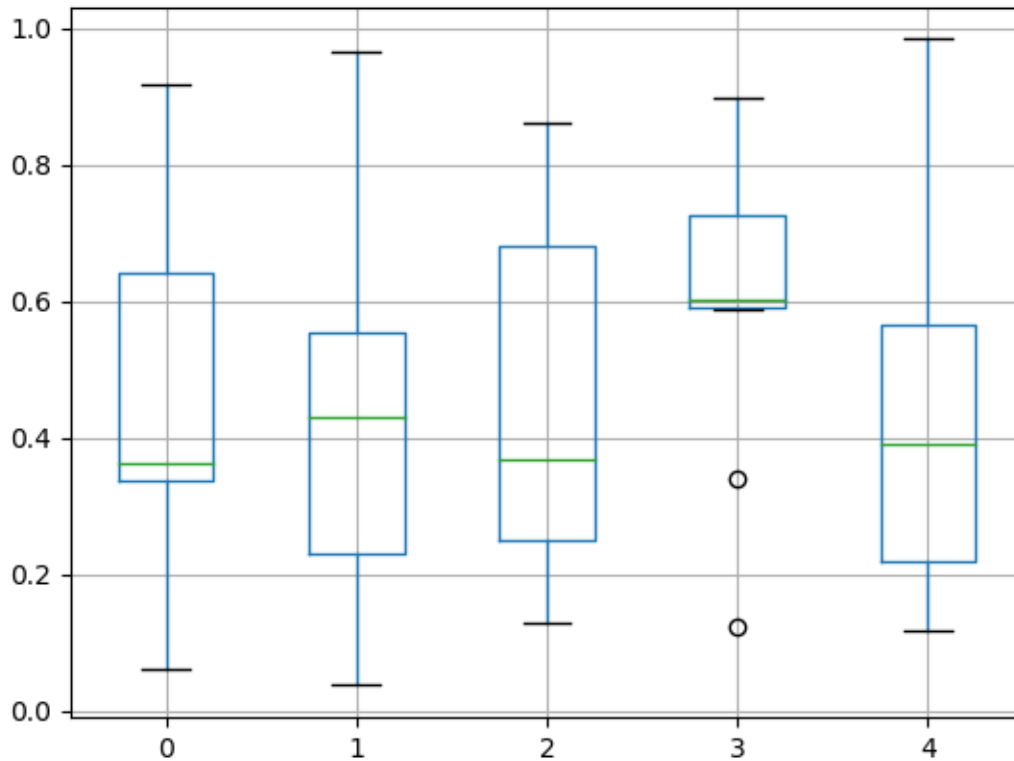
See the [boxplot](#) method and the [matplotlib boxplot documentation](#) for more.

The existing interface `DataFrame.boxplot` to plot boxplot still can be used.

```
In [39]: df = pd.DataFrame(np.random.rand(10, 5))
```

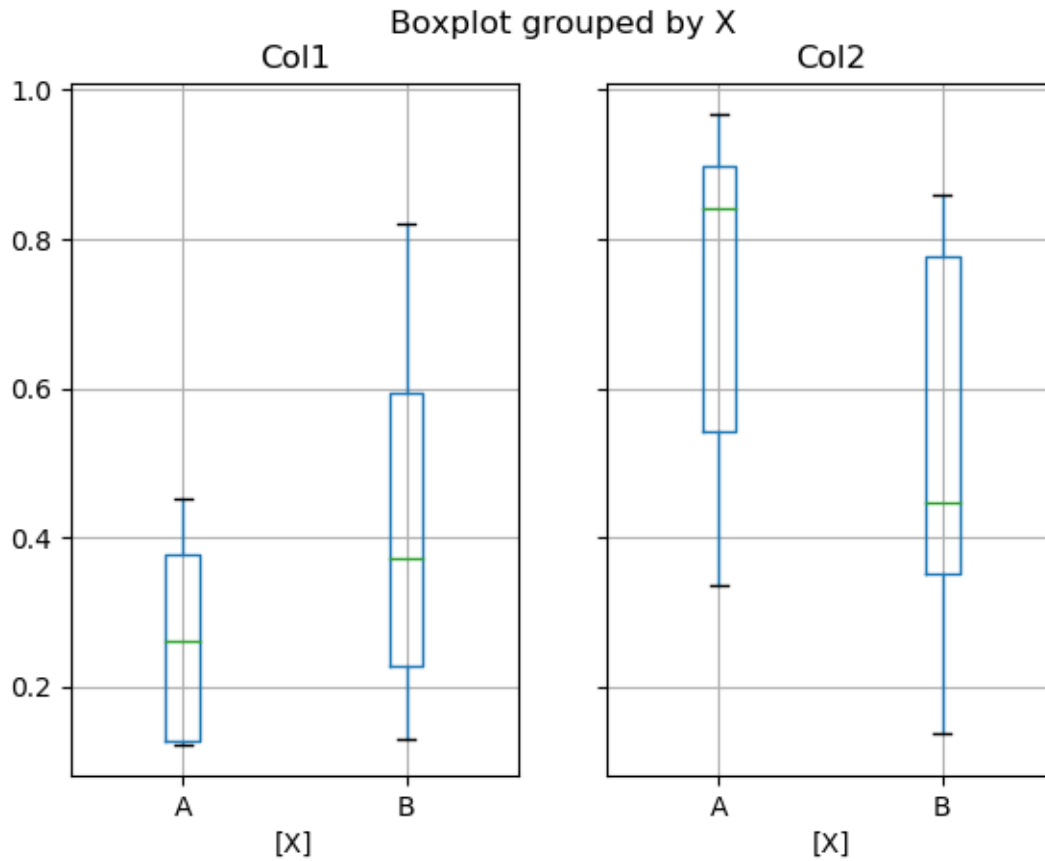
```
In [40]: plt.figure();
```

```
In [41]: bp = df.boxplot()
```



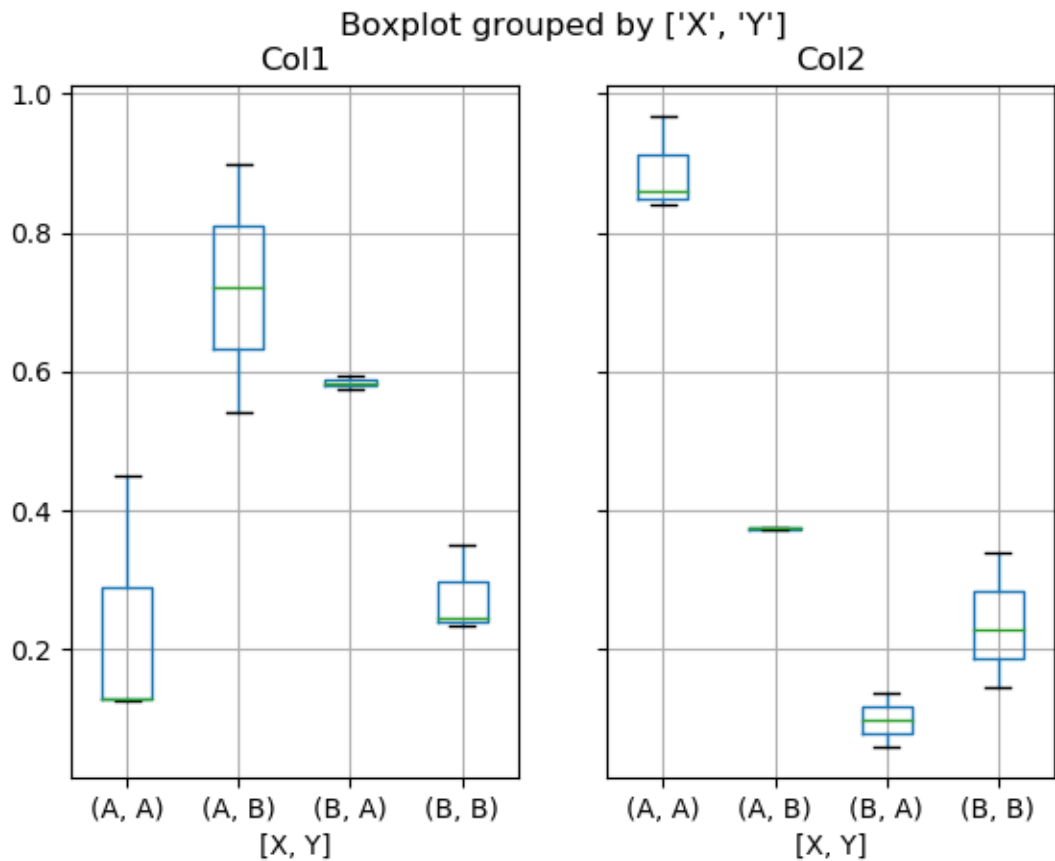
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

```
In [42]: df = pd.DataFrame(np.random.rand(10,2), columns=['Col1', 'Col2'] )
In [43]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [44]: plt.figure();
In [45]: bp = df.boxplot(by='X')
```



You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [46]: df = pd.DataFrame(np.random.rand(10,3), columns=['Col1', 'Col2', 'Col3'])
In [47]: df['X'] = pd.Series(['A','A','A','A','A','B','B','B','B','B'])
In [48]: df['Y'] = pd.Series(['A','B','A','B','A','B','A','B','A','B'])
In [49]: plt.figure();
In [50]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```



**Warning:** The default changed from 'dict' to 'axes' in version 0.19.0.

In `boxplot`, the return type can be controlled by the `return_type`, keyword. The valid choices are {"axes", "dict", "both", None}. Faceting, created by `DataFrame.boxplot` with the `by` keyword, will affect the output type as well:

| return_type= | Faceted | Output type                |
|--------------|---------|----------------------------|
| None         | No      | axes                       |
| None         | Yes     | 2-D ndarray of axes        |
| 'axes'       | No      | axes                       |
| 'axes'       | Yes     | Series of axes             |
| 'dict'       | No      | dict of artists            |
| 'dict'       | Yes     | Series of dicts of artists |
| 'both'       | No      | namedtuple                 |
| 'both'       | Yes     | Series of namedtuples      |

`Groupby.boxplot` always returns a `Series` of `return_type`.

```
In [51]: np.random.seed(1234)
In [52]: df_box = pd.DataFrame(np.random.randn(50, 2))
```

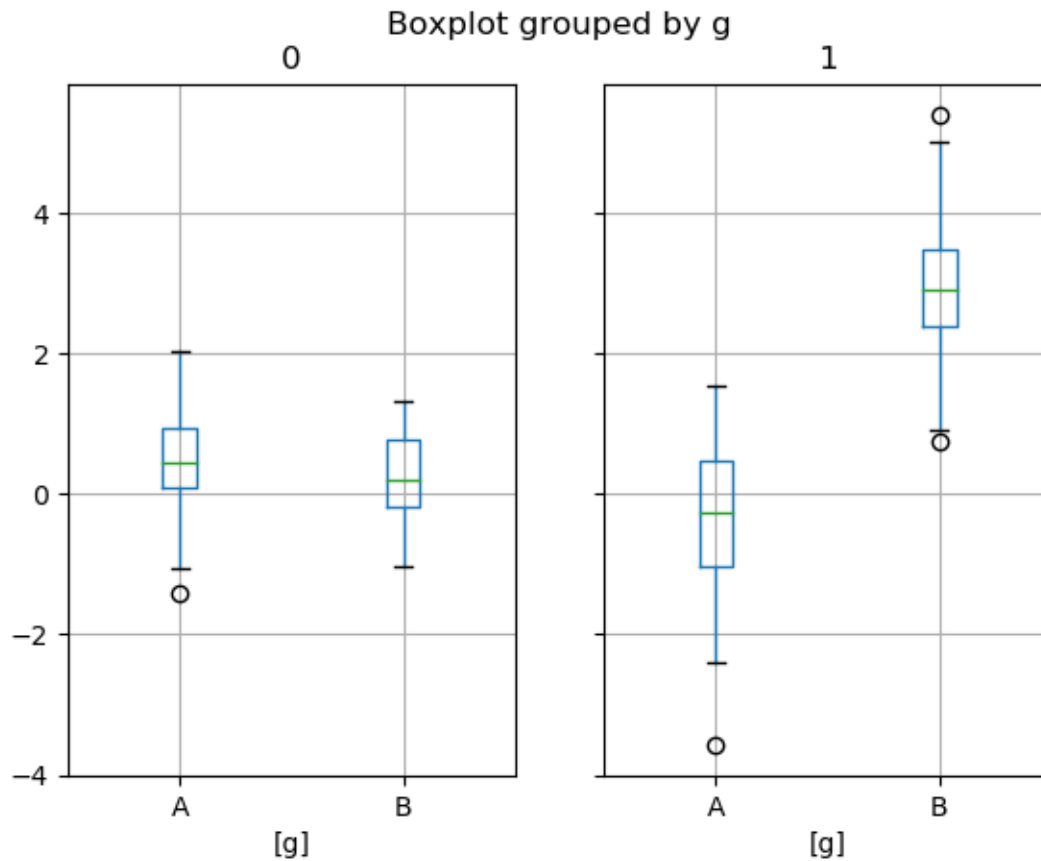
(continues on next page)

(continued from previous page)

```
In [53]: df_box['g'] = np.random.choice(['A', 'B'], size=50)
```

```
In [54]: df_box.loc[df_box['g'] == 'B', 1] += 3
```

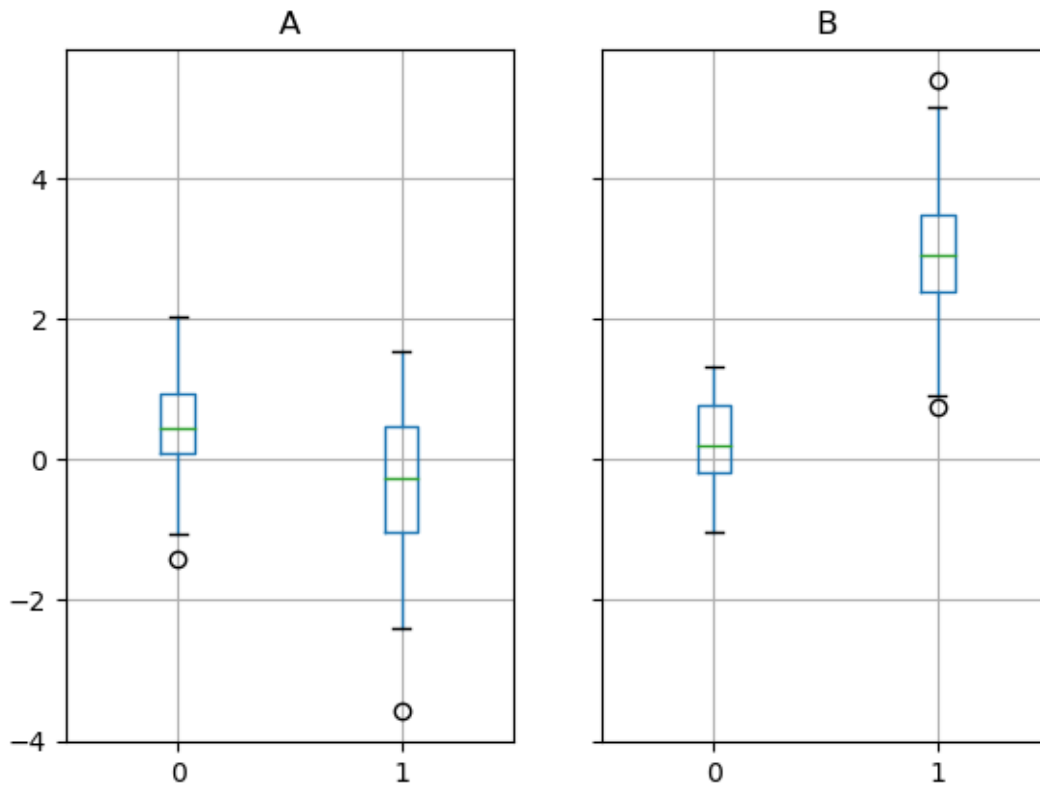
```
In [55]: bp = df_box.boxplot(by='g')
```



The subplots above are split by the numeric columns first, then the value of the `g` column. Below the subplots are first split by the value of `g`, then by the numeric columns.

```
In [56]: bp = df_box.groupby('g').boxplot()
```





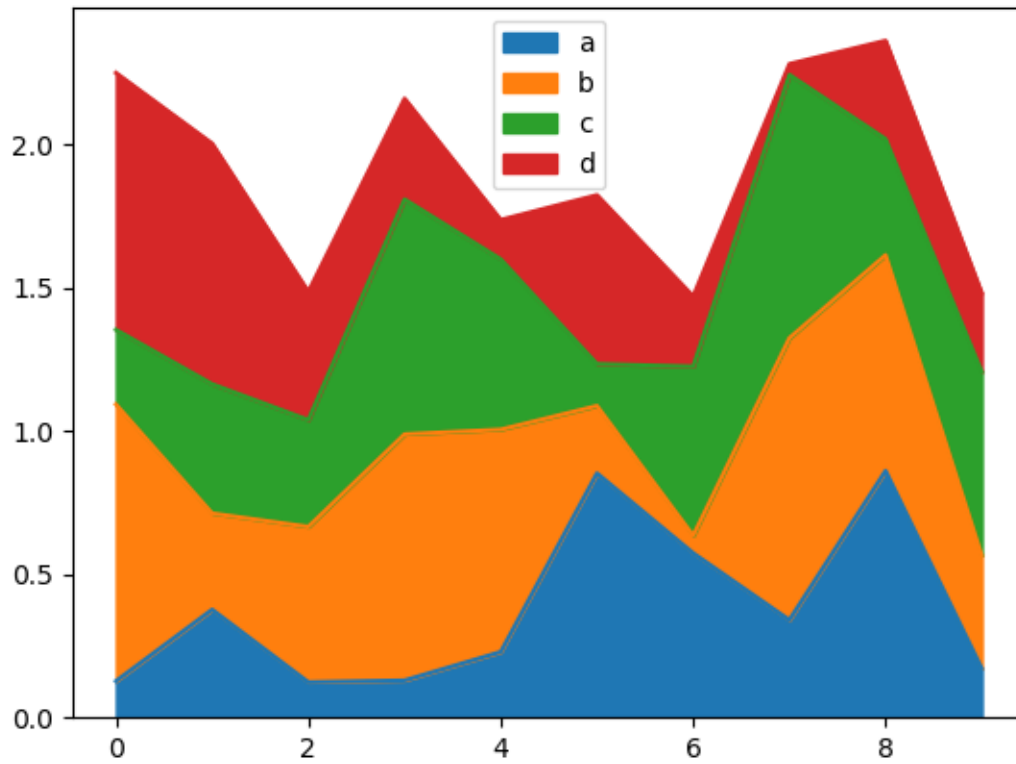
### 22.2.4 Area Plot

You can create area plots with `Series.plot.area()` and `DataFrame.plot.area()`. Area plots are stacked by default. To produce stacked area plot, each column must be either all positive or all negative values.

When input data contains *NaN*, it will be automatically filled by 0. If you want to drop or fill by different values, use `dataframe.dropna()` or `dataframe.fillna()` before calling `plot`.

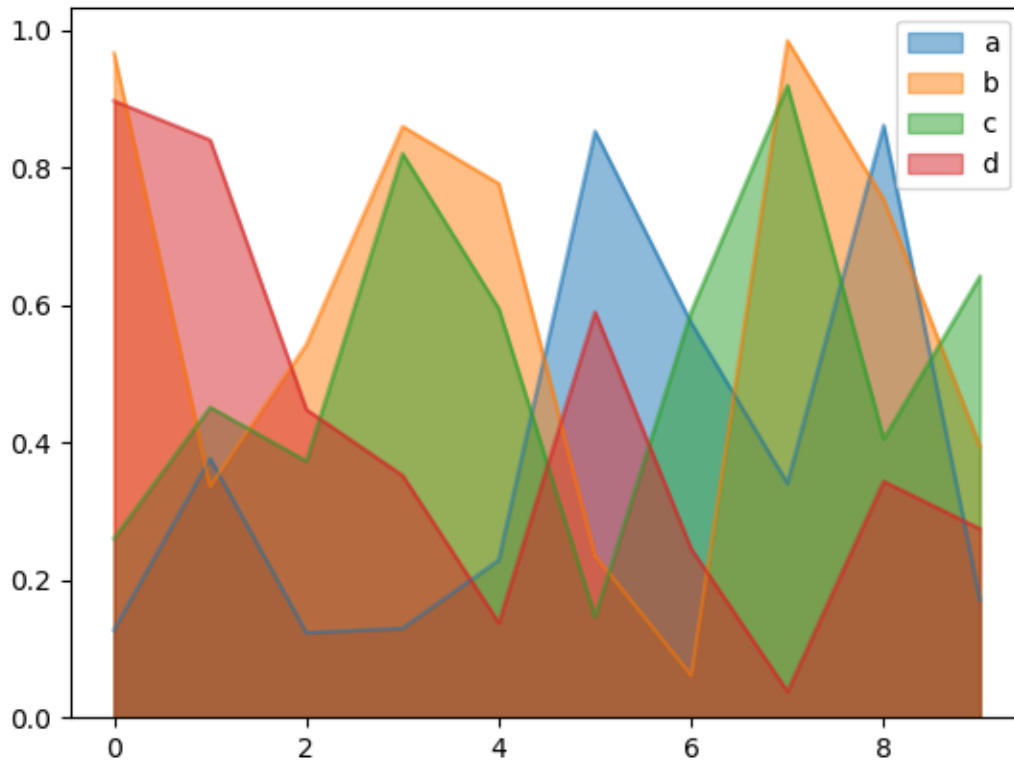
```
In [57]: df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [58]: df.plot.area();
```



To produce an unstacked plot, pass `stacked=False`. Alpha value is set to 0.5 unless otherwise specified:

```
In [59]: df.plot.area(stacked=False);
```

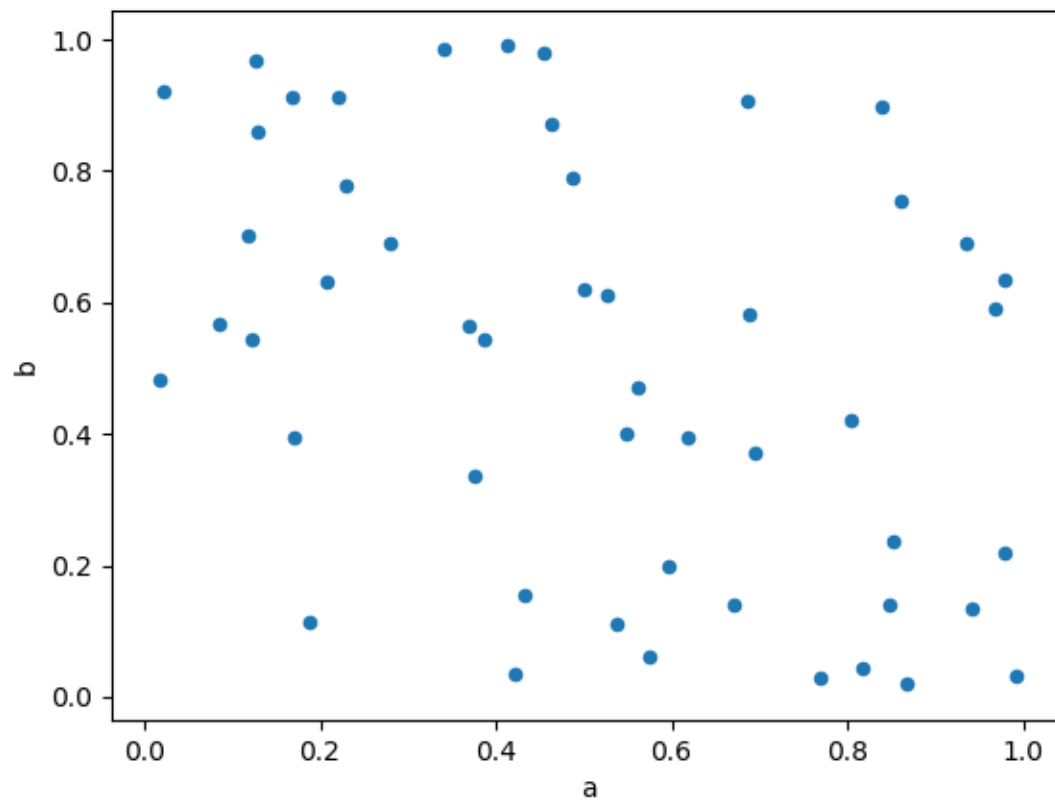


### 22.2.5 Scatter Plot

Scatter plot can be drawn by using the `DataFrame.plot.scatter()` method. Scatter plot requires numeric columns for the x and y axes. These can be specified by the `x` and `y` keywords.

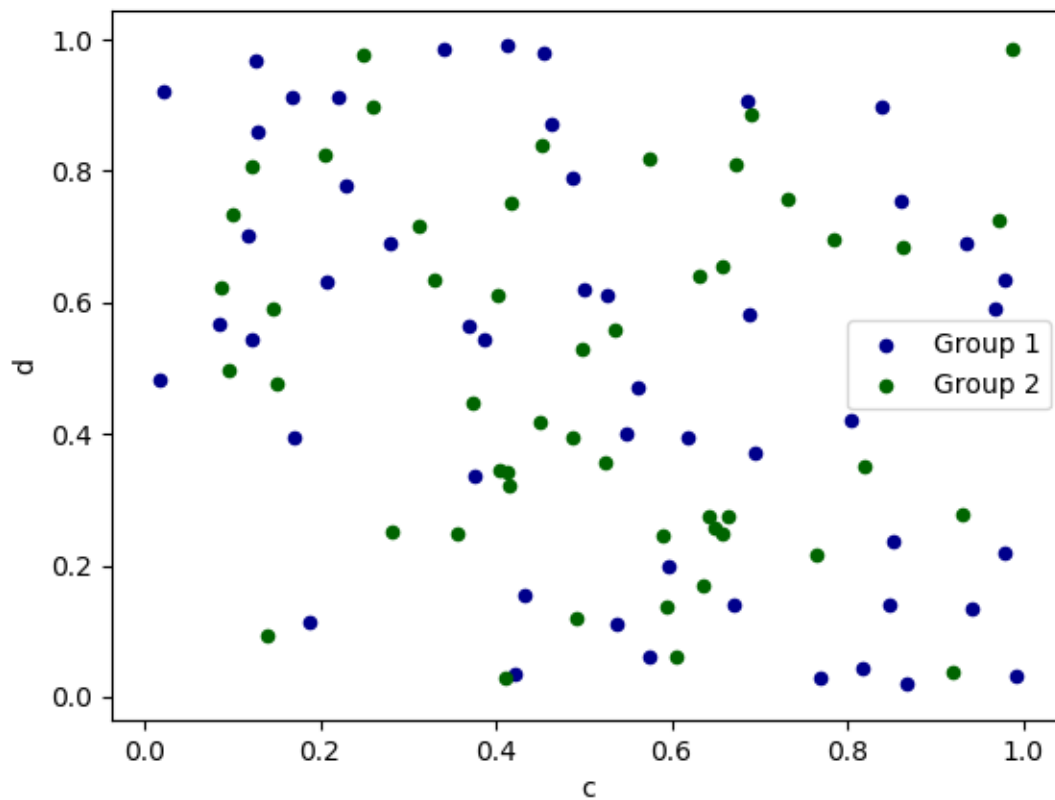
```
In [60]: df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c', 'd'])
```

```
In [61]: df.plot.scatter(x='a', y='b');
```



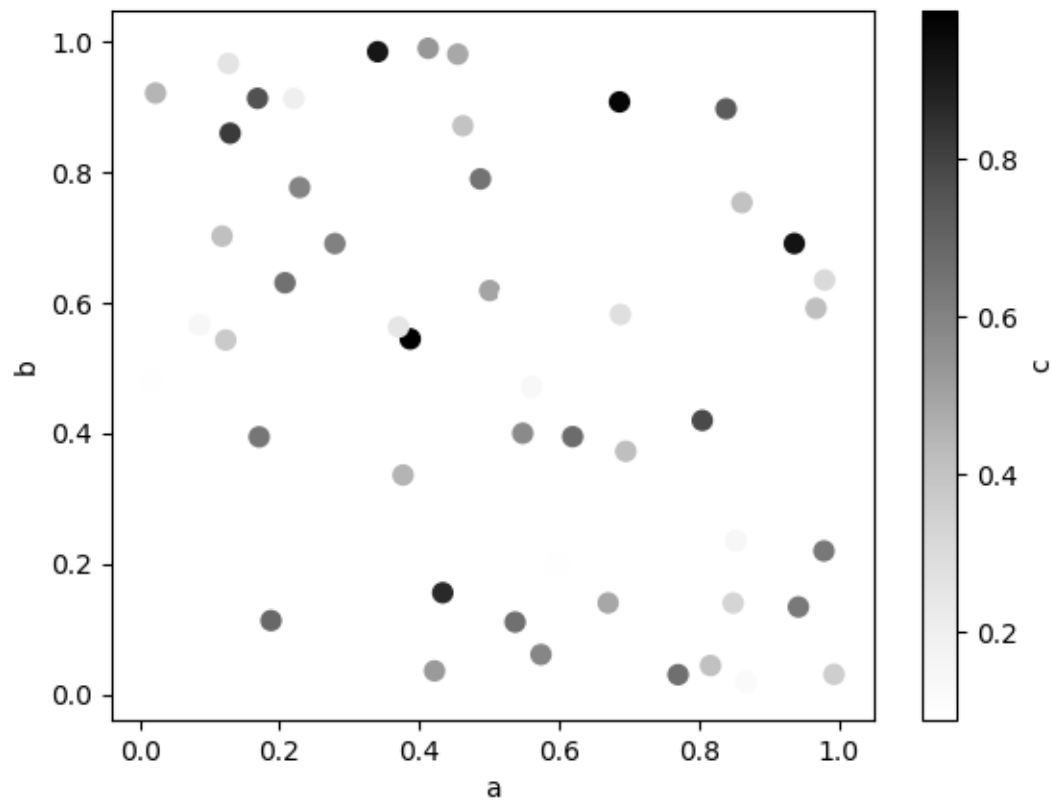
To plot multiple column groups in a single axes, repeat `plot` method specifying target `ax`. It is recommended to specify `color` and `label` keywords to distinguish each groups.

```
In [62]: ax = df.plot.scatter(x='a', y='b', color='DarkBlue', label='Group 1');  
In [63]: df.plot.scatter(x='c', y='d', color='DarkGreen', label='Group 2', ax=ax);
```



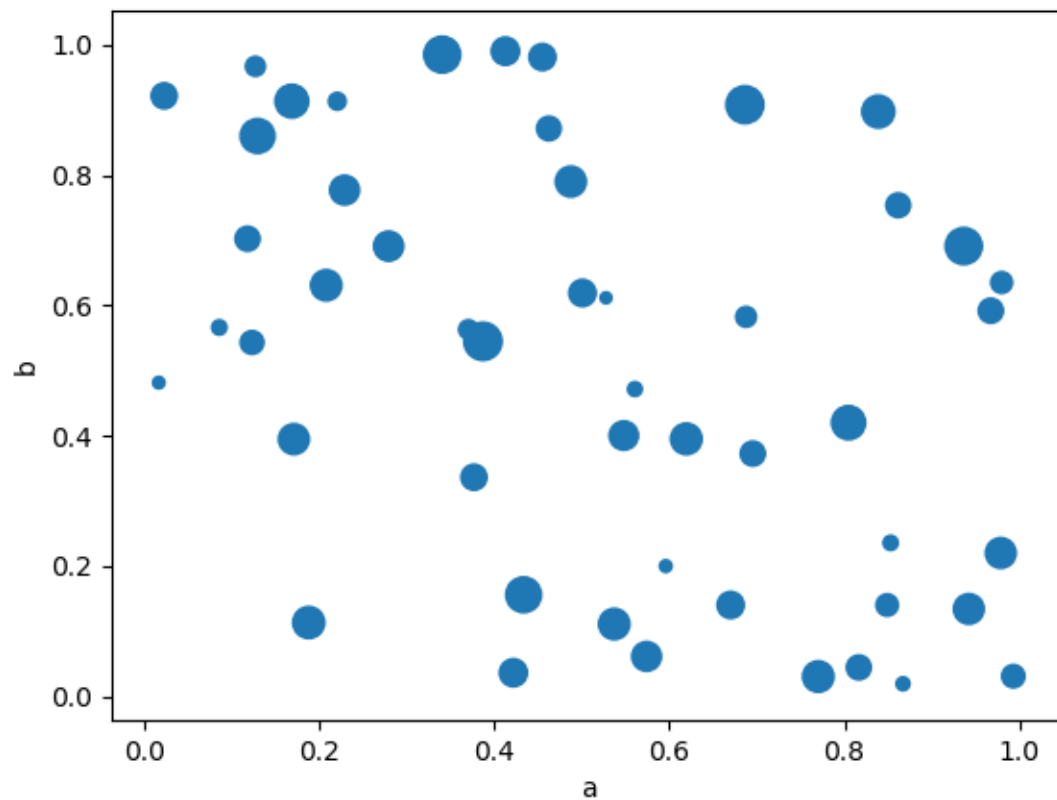
The keyword `c` may be given as the name of a column to provide colors for each point:

```
In [64]: df.plot.scatter(x='a', y='b', c='c', s=50);
```



You can pass other keywords supported by matplotlib `scatter`. The example below shows a bubble chart using a column of the `DataFrame` as the bubble size.

```
In [65]: df.plot.scatter(x='a', y='b', s=df['c']*200);
```

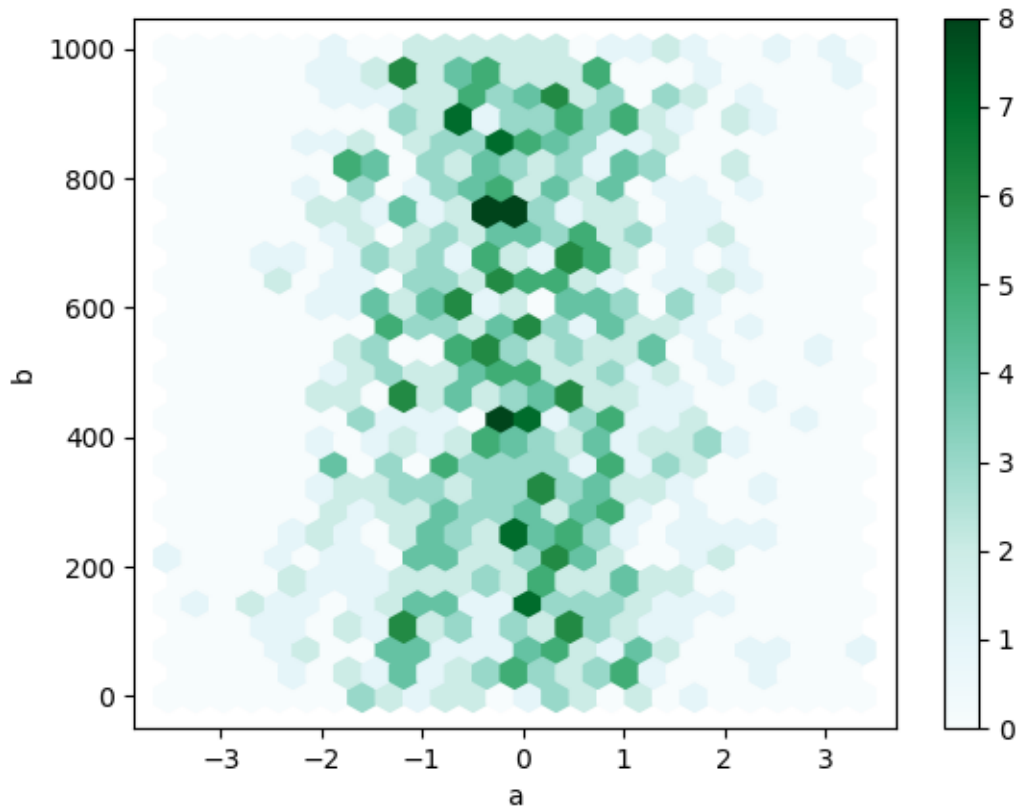


See the `scatter` method and the [matplotlib scatter documentation](#) for more.

### 22.2.6 Hexagonal Bin Plot

You can create hexagonal bin plots with `DataFrame.plot.hexbin()`. Hexbin plots can be a useful alternative to scatter plots if your data are too dense to plot each point individually.

```
In [66]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])  
In [67]: df['b'] = df['b'] + np.arange(1000)  
In [68]: df.plot.hexbin(x='a', y='b', gridsize=25)  
Out[68]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3d4c09b0>
```



A useful keyword argument is `gridsize`; it controls the number of hexagons in the x-direction, and defaults to 100. A larger `gridsize` means more, smaller bins.

By default, a histogram of the counts around each  $(x, y)$  point is computed. You can specify alternative aggregations by passing values to the `C` and `reduce_C_function` arguments. `C` specifies the value at each  $(x, y)$  point and `reduce_C_function` is a function of one argument that reduces all the values in a bin to a single number (e.g. mean, max, sum, std). In this example the positions are given by columns `a` and `b`, while the value is given by column `z`. The bins are aggregated with NumPy's `max` function.

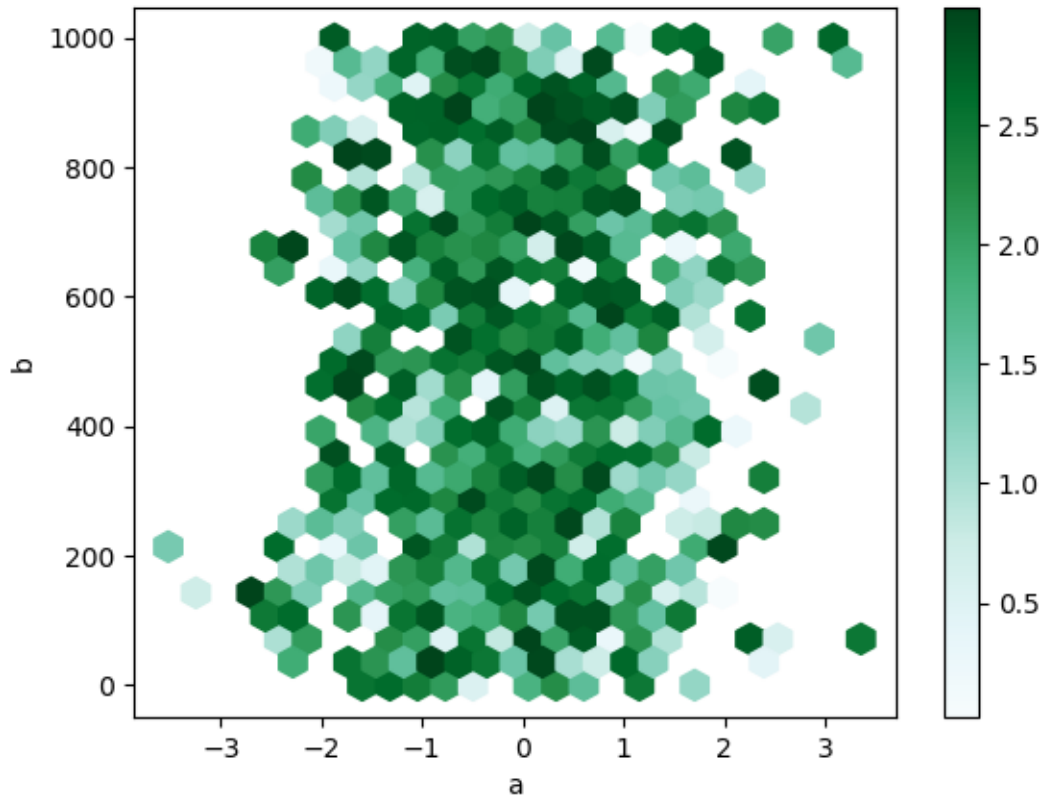
```
In [69]: df = pd.DataFrame(np.random.randn(1000, 2), columns=['a', 'b'])

In [70]: df['b'] = df['b'] + np.arange(1000)

In [71]: df['z'] = np.random.uniform(0, 3, 1000)

In [72]: df.plot.hexbin(x='a', y='b', C='z', reduce_C_function=np.max,
.....:                 gridsize=25)
.....:
Out[72]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3d567dd8>
```





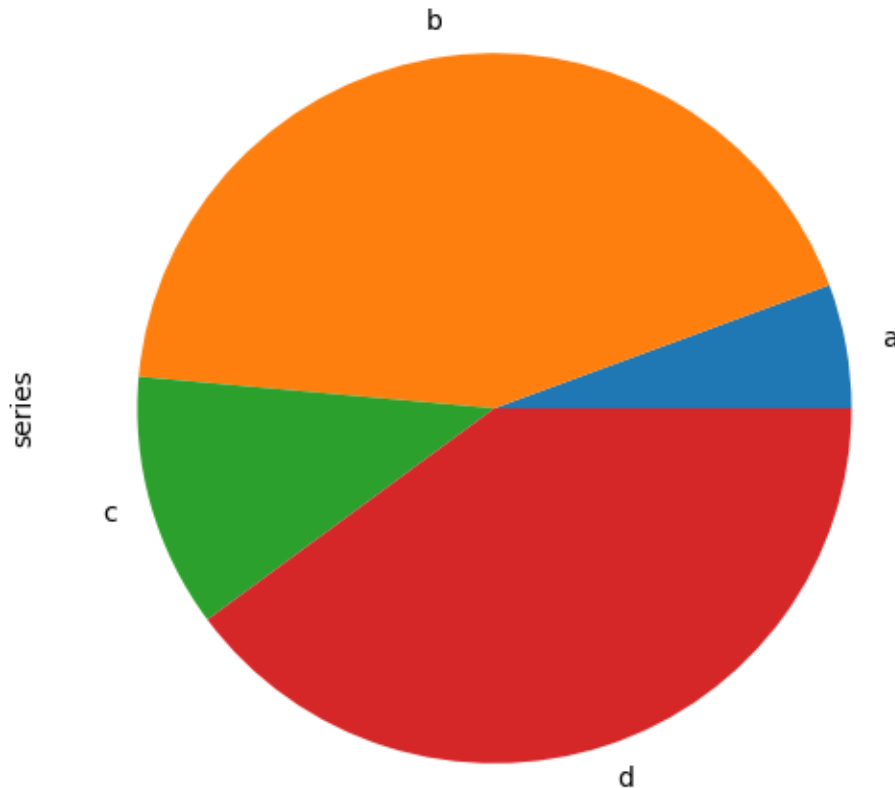
See the `hexbin` method and the `matplotlib hexbin` documentation for more.

### 22.2.7 Pie plot

You can create a pie plot with `DataFrame.plot.pie()` or `Series.plot.pie()`. If your data includes any `NaN`, they will be automatically filled with 0. A `ValueError` will be raised if there are any negative values in your data.

```
In [73]: series = pd.Series(3 * np.random.rand(4), index=['a', 'b', 'c', 'd'], name=
↳ 'series')

In [74]: series.plot.pie(figsize=(6, 6))
Out [74]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3d85c5f8>
```

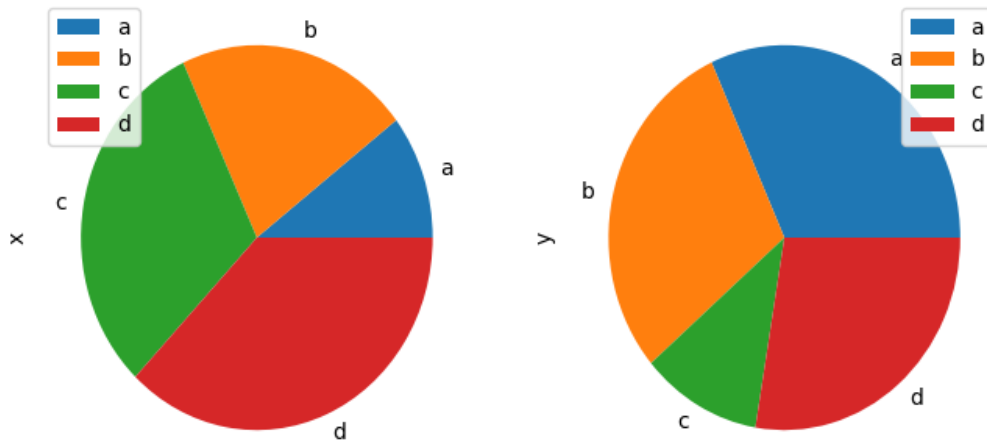


For pie plots it's best to use square figures, i.e. a figure aspect ratio 1. You can create the figure with equal width and height, or force the aspect ratio to be equal after plotting by calling `ax.set_aspect('equal')` on the returned axes object.

Note that pie plot with `DataFrame` requires that you either specify a target column by the `y` argument or `subplots=True`. When `y` is specified, pie plot of selected column will be drawn. If `subplots=True` is specified, pie plots for each column are drawn as subplots. A legend will be drawn in each pie plots by default; specify `legend=False` to hide it.

```
In [75]: df = pd.DataFrame(3 * np.random.rand(4, 2), index=['a', 'b', 'c', 'd'],
↳ columns=['x', 'y'])

In [76]: df.plot.pie(subplots=True, figsize=(8, 4))
Out[76]: array([<matplotlib.axes._subplots.AxesSubplot object at 0x1c3d8c5940>,
<matplotlib.axes._subplots.AxesSubplot object at 0x1c3da4bd30>], dtype=object)
```

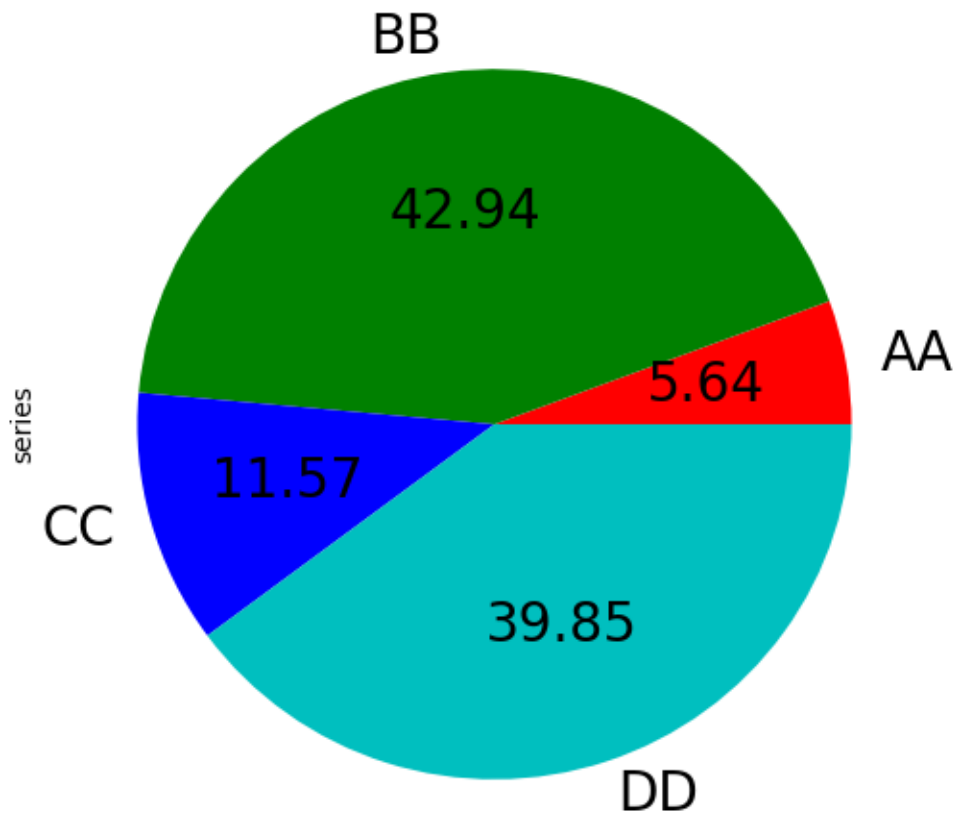


You can use the `labels` and `colors` keywords to specify the labels and colors of each wedge.

**Warning:** Most pandas plots use the `label` and `color` arguments (note the lack of “s” on those). To be consistent with `matplotlib.pyplot.pie()` you must use `labels` and `colors`.

If you want to hide wedge labels, specify `labels=None`. If `fontsize` is specified, the value will be applied to wedge labels. Also, other keywords supported by `matplotlib.pyplot.pie()` can be used.

```
In [77]: series.plot.pie(labels=['AA', 'BB', 'CC', 'DD'], colors=['r', 'g', 'b', 'c'],
.....:                  autopct='%0.2f', fontsize=20, figsize=(6, 6))
.....:
Out [77]: <matplotlib.axes._subplots.AxesSubplot at 0x121551ba8>
```

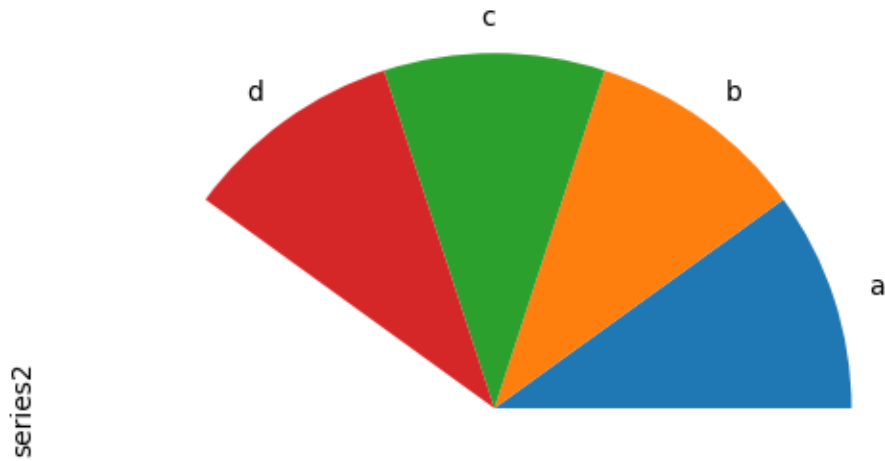


If you pass values whose sum total is less than 1.0, matplotlib draws a semicircle.

```
In [78]: series = pd.Series([0.1] * 4, index=['a', 'b', 'c', 'd'], name='series2')
```

```
In [79]: series.plot.pie(figsize=(6, 6))
```

```
Out [79]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3d8b22b0>
```



See the [matplotlib pie documentation](#) for more.

## 22.3 Plotting with Missing Data

Pandas tries to be pragmatic about plotting `DataFrames` or `Series` that contain missing data. Missing values are dropped, left out, or filled depending on the plot type.

| Plot Type      | NaN Handling            |
|----------------|-------------------------|
| Line           | Leave gaps at NaNs      |
| Line (stacked) | Fill 0's                |
| Bar            | Fill 0's                |
| Scatter        | Drop NaNs               |
| Histogram      | Drop NaNs (column-wise) |
| Box            | Drop NaNs (column-wise) |
| Area           | Fill 0's                |
| KDE            | Drop NaNs (column-wise) |
| Hexbin         | Drop NaNs               |
| Pie            | Fill 0's                |

If any of these defaults are not what you want, or if you want to be explicit about how missing values are handled, consider using `fillna()` or `dropna()` before plotting.

## 22.4 Plotting Tools

These functions can be imported from `pandas.plotting` and take a *Series* or *DataFrame* as an argument.

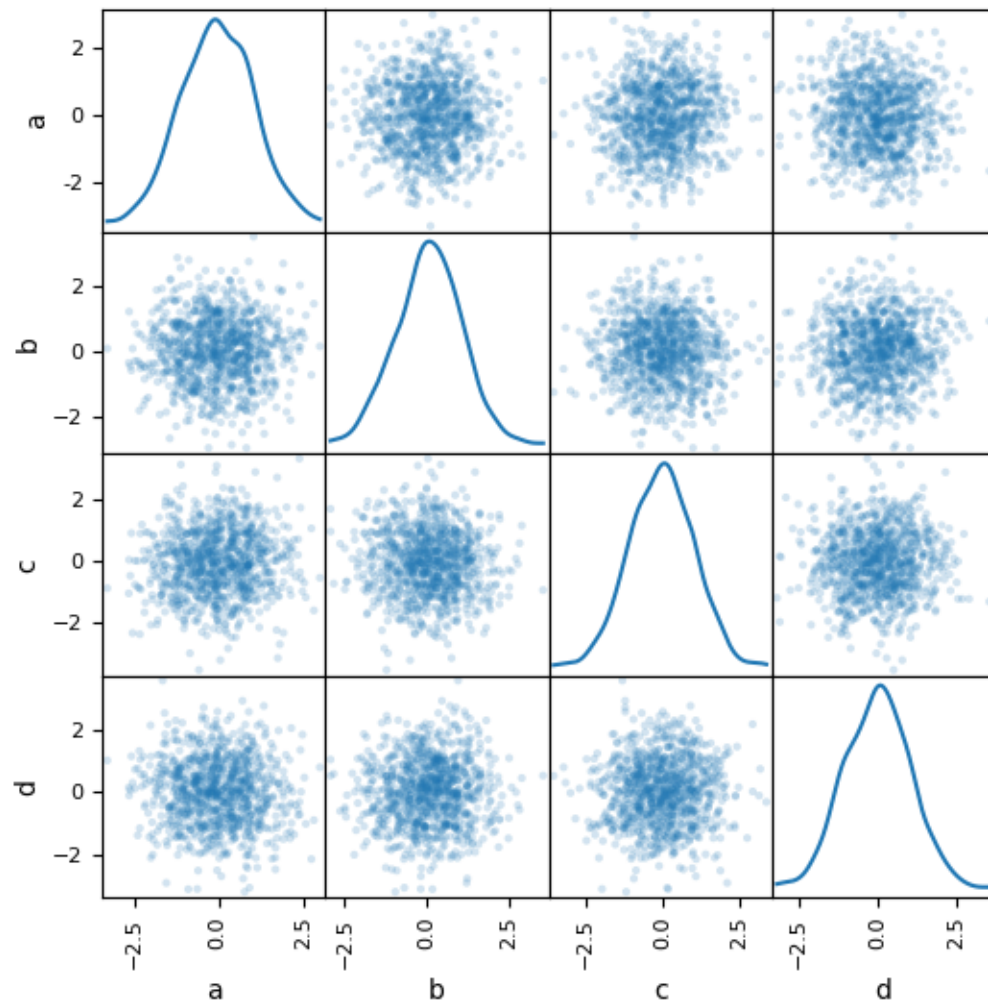
### 22.4.1 Scatter Matrix Plot

You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.plotting`:

```
In [80]: from pandas.plotting import scatter_matrix

In [81]: df = pd.DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])

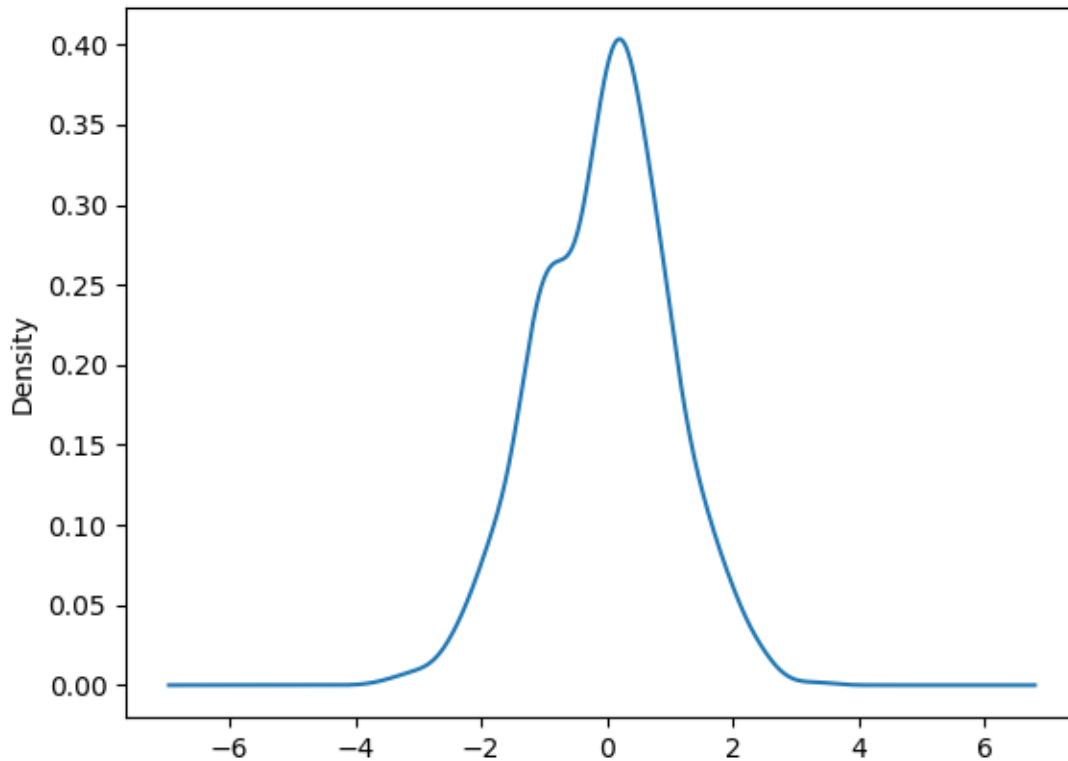
In [82]: scatter_matrix(df, alpha=0.2, figsize=(6, 6), diagonal='kde')
Out[82]:
array([[<matplotlib.axes._subplots.AxesSubplot object at 0x1c381e89e8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c37c4da20>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c3781c898>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c376d50f0>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x1c33b56320>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c33b56518>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c2f046550>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c2bdcf780>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x1c370c6400>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c25a6a710>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c2f082a20>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c36ac3668>],
        [<matplotlib.axes._subplots.AxesSubplot object at 0x1c33c43cf8>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c37d01cc0>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c383a0358>,
        <matplotlib.axes._subplots.AxesSubplot object at 0x1c38115470>]],
        dtype=object)
```



### 22.4.2 Density Plot

You can create density plots using the `Series.plot.kde()` and `DataFrame.plot.kde()` methods.

```
In [83]: ser = pd.Series(np.random.randn(1000))  
  
In [84]: ser.plot.kde()  
Out[84]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3842c908>
```



### 22.4.3 Andrews Curves

Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series, see the [Wikipedia entry](#) for more information. By coloring these curves differently for each class it is possible to visualize data clustering. Curves belonging to samples of the same class will usually be closer together and form larger structures.

**Note:** The “Iris” dataset is available [here](#).

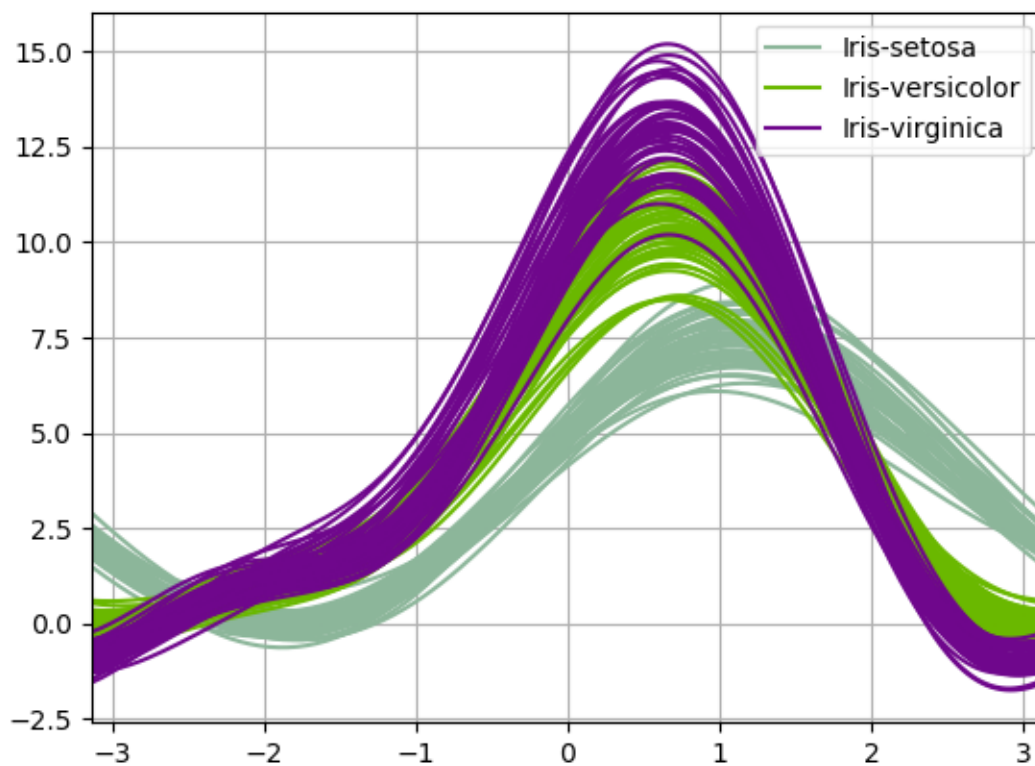
```
In [85]: from pandas.plotting import andrews_curves

In [86]: data = pd.read_csv('data/iris.data')

In [87]: plt.figure()
Out[87]: <Figure size 640x480 with 0 Axes>

In [88]: andrews_curves(data, 'Name')
Out[88]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c38f27da0>
```





#### 22.4.4 Parallel Coordinates

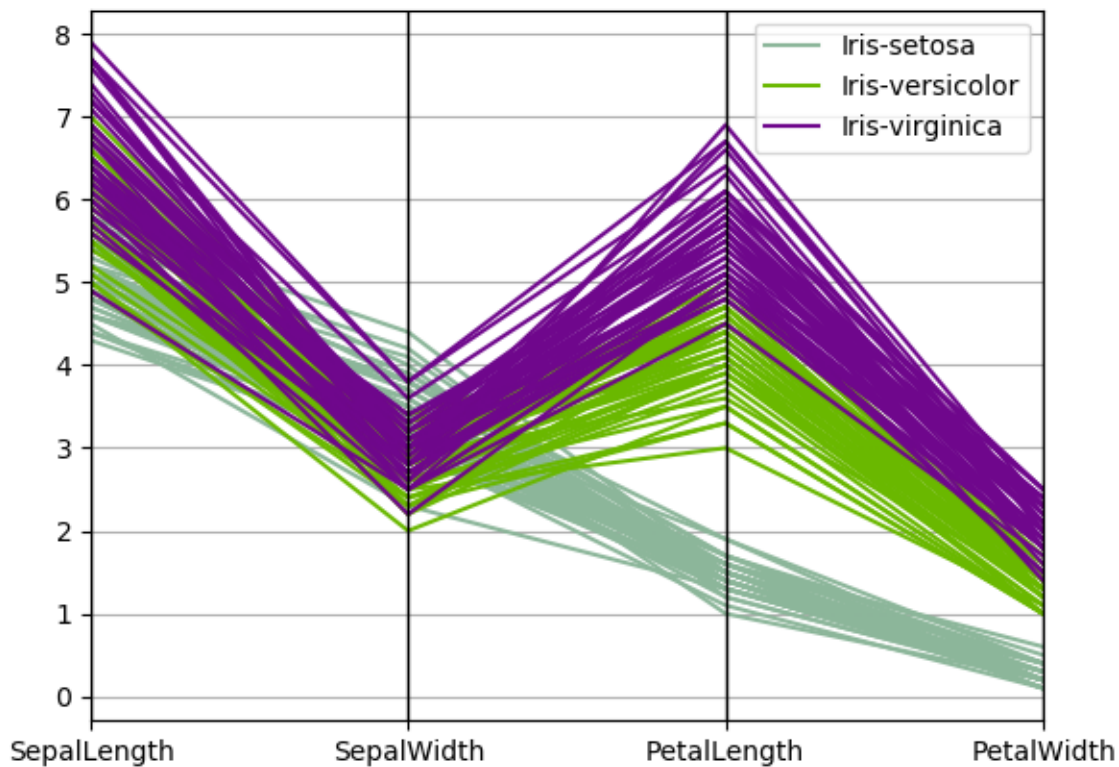
Parallel coordinates is a plotting technique for plotting multivariate data, see the [Wikipedia entry](#) for an introduction. Parallel coordinates allows one to see clusters in data and to estimate other statistics visually. Using parallel coordinates points are represented as connected line segments. Each vertical line represents one attribute. One set of connected line segments represents one data point. Points that tend to cluster will appear closer together.

```
In [89]: from pandas.plotting import parallel_coordinates

In [90]: data = pd.read_csv('data/iris.data')

In [91]: plt.figure()
Out[91]: <Figure size 640x480 with 0 Axes>

In [92]: parallel_coordinates(data, 'Name')
Out[92]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c38e62b38>
```



### 22.4.5 Lag Plot

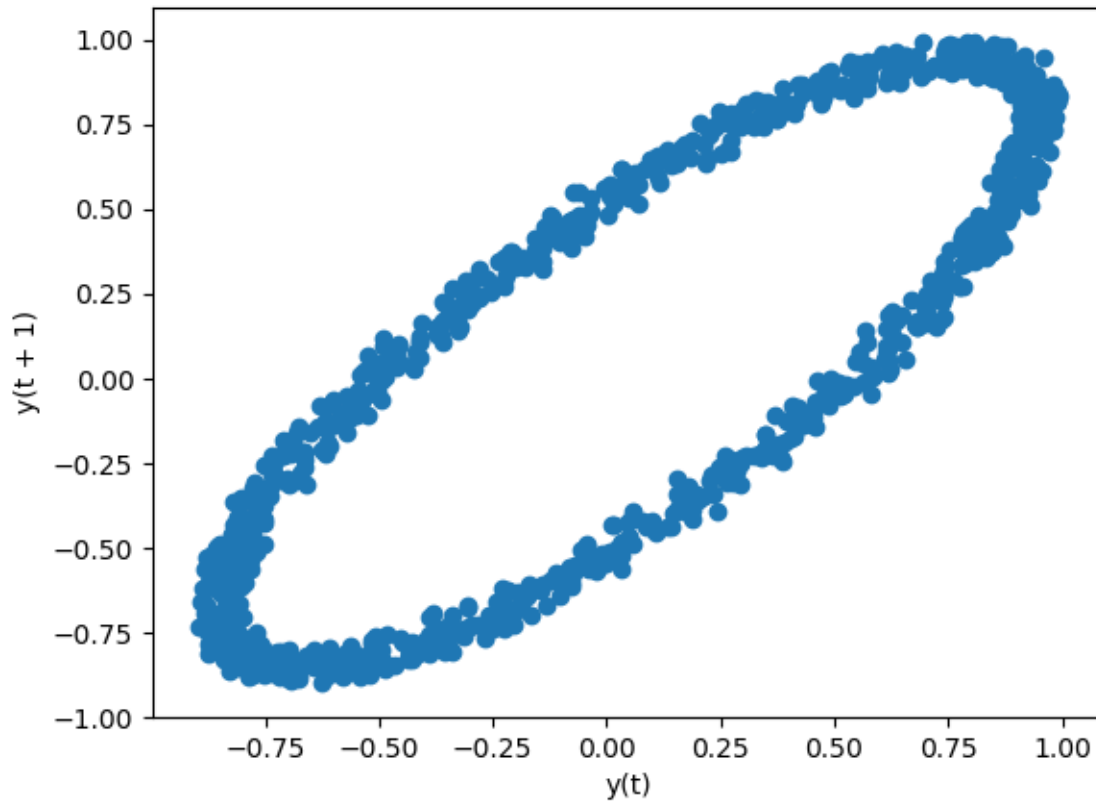
Lag plots are used to check if a data set or time series is random. Random data should not exhibit any structure in the lag plot. Non-random structure implies that the underlying data are not random. The `lag` argument may be passed, and when `lag=1` the plot is essentially `data[:-1]` vs. `data[1:]`.

```
In [93]: from pandas.plotting import lag_plot

In [94]: plt.figure()
Out[94]: <Figure size 640x480 with 0 Axes>

In [95]: data = pd.Series(0.1 * np.random.rand(1000) +
.....:     0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))
.....:

In [96]: lag_plot(data)
Out[96]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2b26e208>
```



### 22.4.6 Autocorrelation Plot

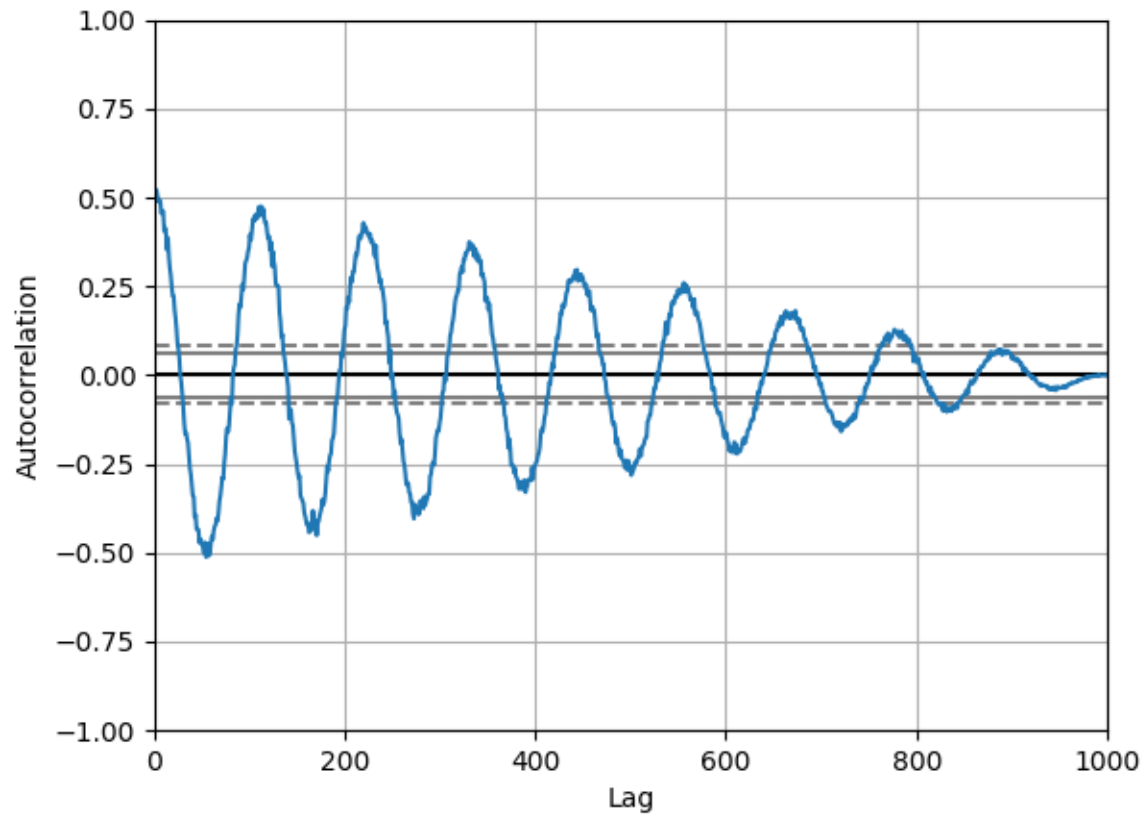
Autocorrelation plots are often used for checking randomness in time series. This is done by computing autocorrelations for data values at varying time lags. If time series is random, such autocorrelations should be near zero for any and all time-lag separations. If time series is non-random then one or more of the autocorrelations will be significantly non-zero. The horizontal lines displayed in the plot correspond to 95% and 99% confidence bands. The dashed line is 99% confidence band. See the [Wikipedia entry](#) for more about autocorrelation plots.

```
In [97]: from pandas.plotting import autocorrelation_plot

In [98]: plt.figure()
Out[98]: <Figure size 640x480 with 0 Axes>

In [99]: data = pd.Series(0.7 * np.random.rand(1000) +
.....:     0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
.....:

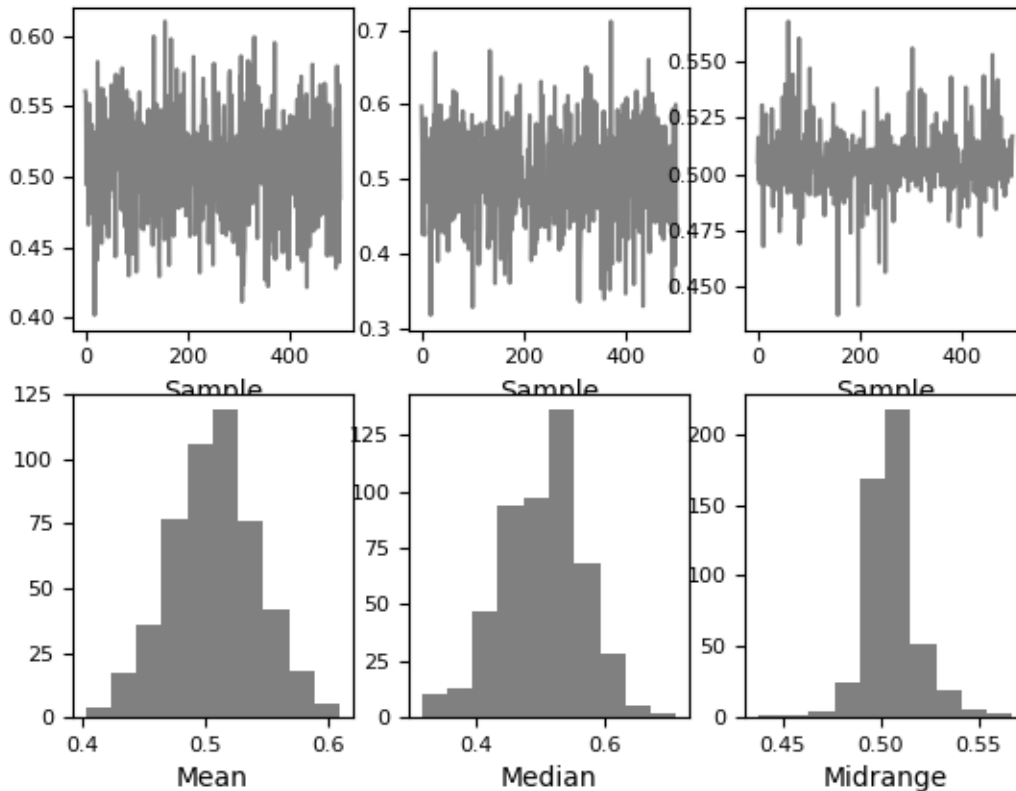
In [100]: autocorrelation_plot(data)
Out[100]: <matplotlib.axes._subplots.AxesSubplot at 0x1c37dd3588>
```



### 22.4.7 Bootstrap Plot

Bootstrap plots are used to visually assess the uncertainty of a statistic, such as mean, median, midrange, etc. A random subset of a specified size is selected from a data set, the statistic in question is computed for this subset and the process is repeated a specified number of times. Resulting plots and histograms are what constitutes the bootstrap plot.

```
In [101]: from pandas.plotting import bootstrap_plot
In [102]: data = pd.Series(np.random.rand(1000))
In [103]: bootstrap_plot(data, size=50, samples=500, color='grey')
Out[103]: <Figure size 640x480 with 6 Axes>
```



## 22.4.8 RadViz

RadViz is a way of visualizing multi-variate data. It is based on a simple spring tension minimization algorithm. Basically you set up a bunch of points in a plane. In our case they are equally spaced on a unit circle. Each point represents a single attribute. You then pretend that each sample in the data set is attached to each of these points by a spring, the stiffness of which is proportional to the numerical value of that attribute (they are normalized to unit interval). The point in the plane, where our sample settles to (where the forces acting on our sample are at an equilibrium) is where a dot representing our sample will be drawn. Depending on which class that sample belongs to it will be colored differently. See the R package [Radviz](#) for more information.

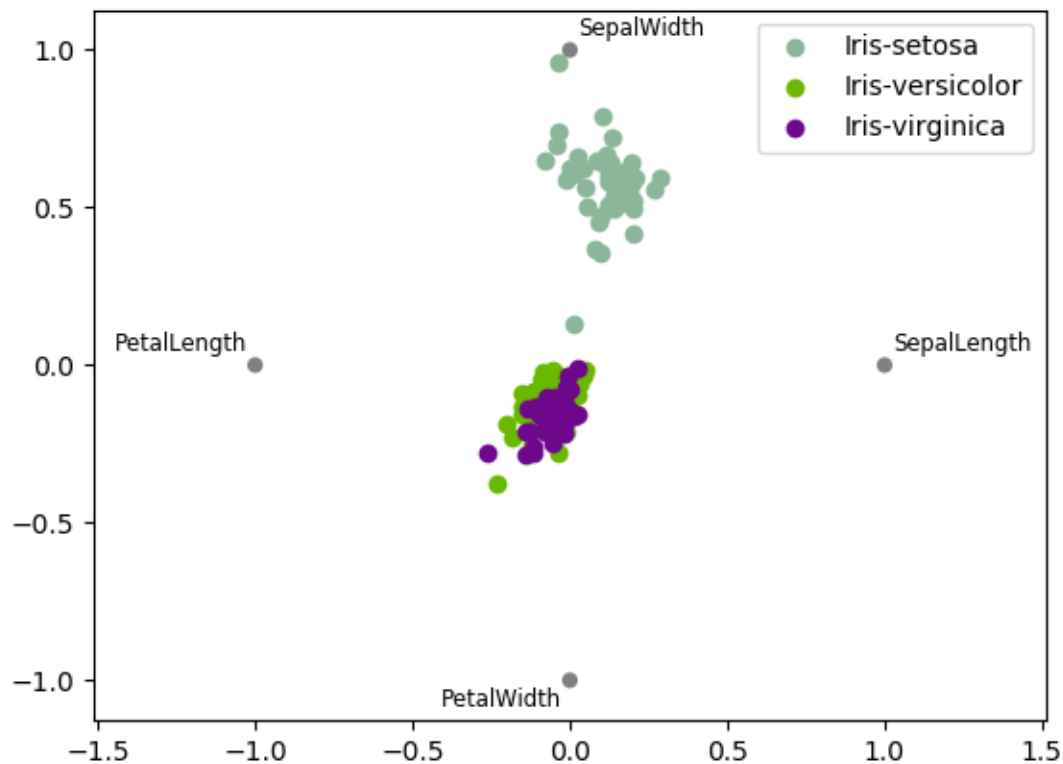
**Note:** The “Iris” dataset is available [here](#).

```
In [104]: from pandas.plotting import radviz

In [105]: data = pd.read_csv('data/iris.data')

In [106]: plt.figure()
Out[106]: <Figure size 640x480 with 0 Axes>

In [107]: radviz(data, 'Name')
Out[107]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c3747f470>
```



## 22.5 Plot Formatting

### 22.5.1 Setting the plot style

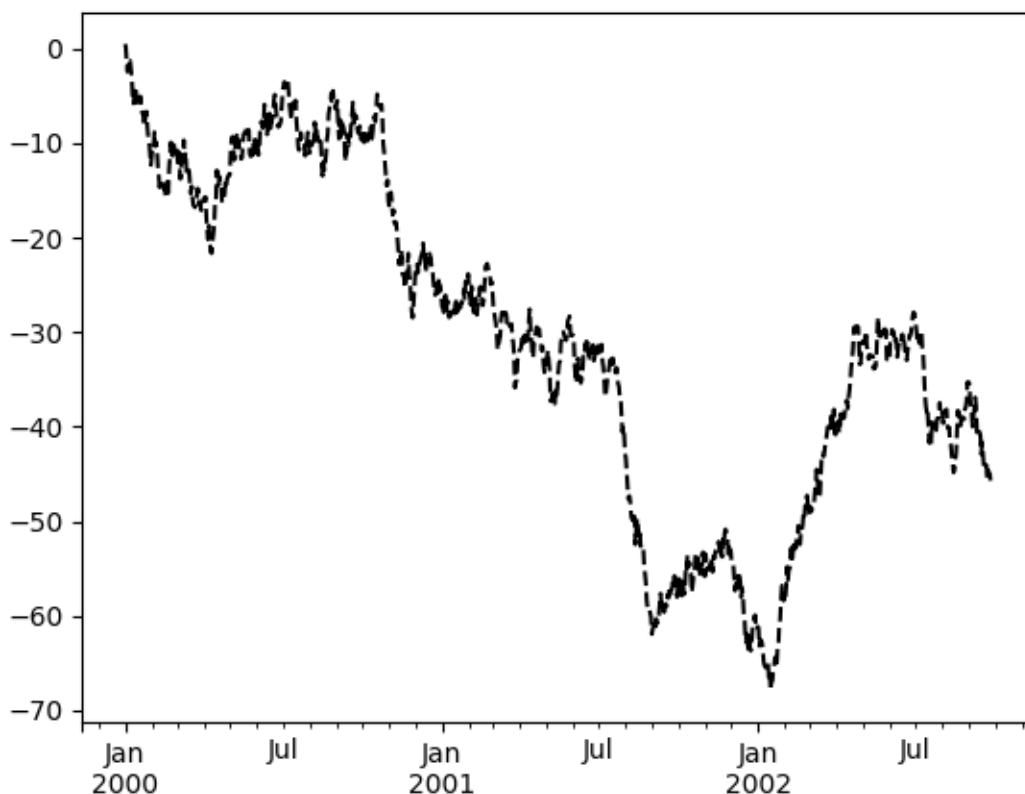
From version 1.5 and up, matplotlib offers a range of preconfigured plotting styles. Setting the style can be used to easily give plots the general look that you want. Setting the style is as easy as calling `matplotlib.style.use(my_plot_style)` before creating your plot. For example you could write `matplotlib.style.use('ggplot')` for ggplot-style plots.

You can see the various available style names at `matplotlib.style.available` and it's very easy to try them out.

### 22.5.2 General plot style arguments

Most plotting methods have a set of keyword arguments that control the layout and formatting of the returned plot:

```
In [108]: plt.figure(); ts.plot(style='k--', label='Series');
```



For each kind of plot (e.g. *line*, *bar*, *scatter*) any additional arguments keywords are passed along to the corresponding matplotlib function (`ax.plot()`, `ax.bar()`, `ax.scatter()`). These can be used to control additional styling, beyond what pandas provides.

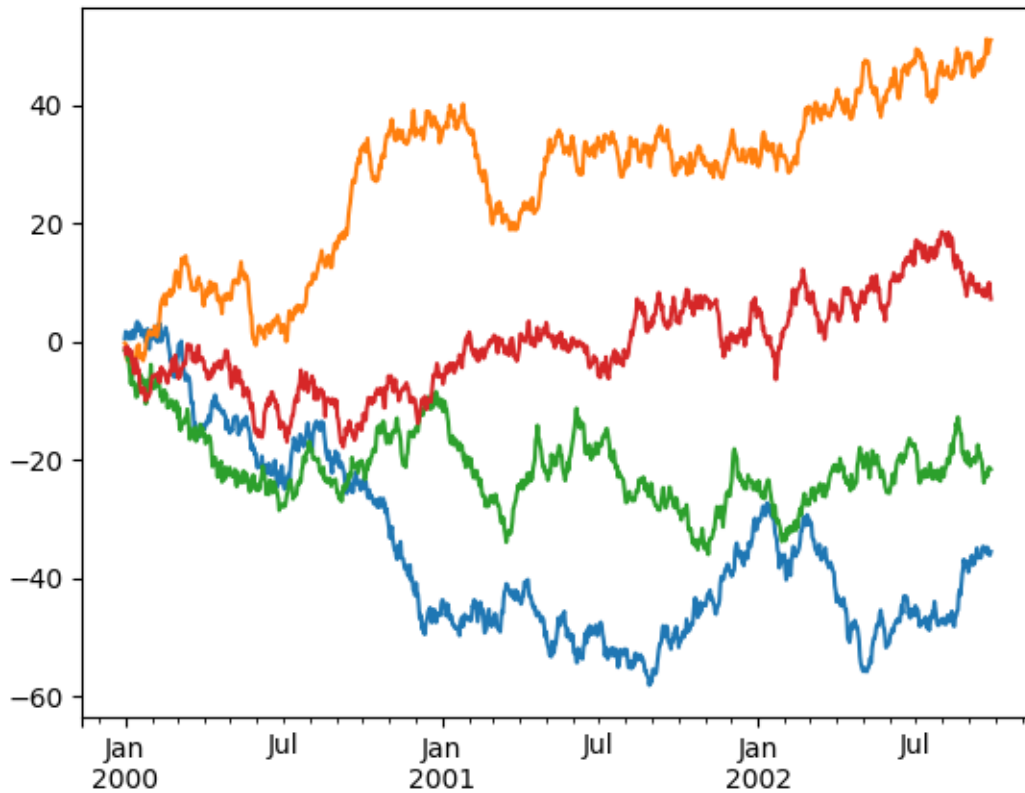
### 22.5.3 Controlling the Legend

You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```
In [109]: df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list(
↳ 'ABCD'))

In [110]: df = df.cumsum()

In [111]: df.plot(legend=False)
Out[111]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2a823fd0>
```

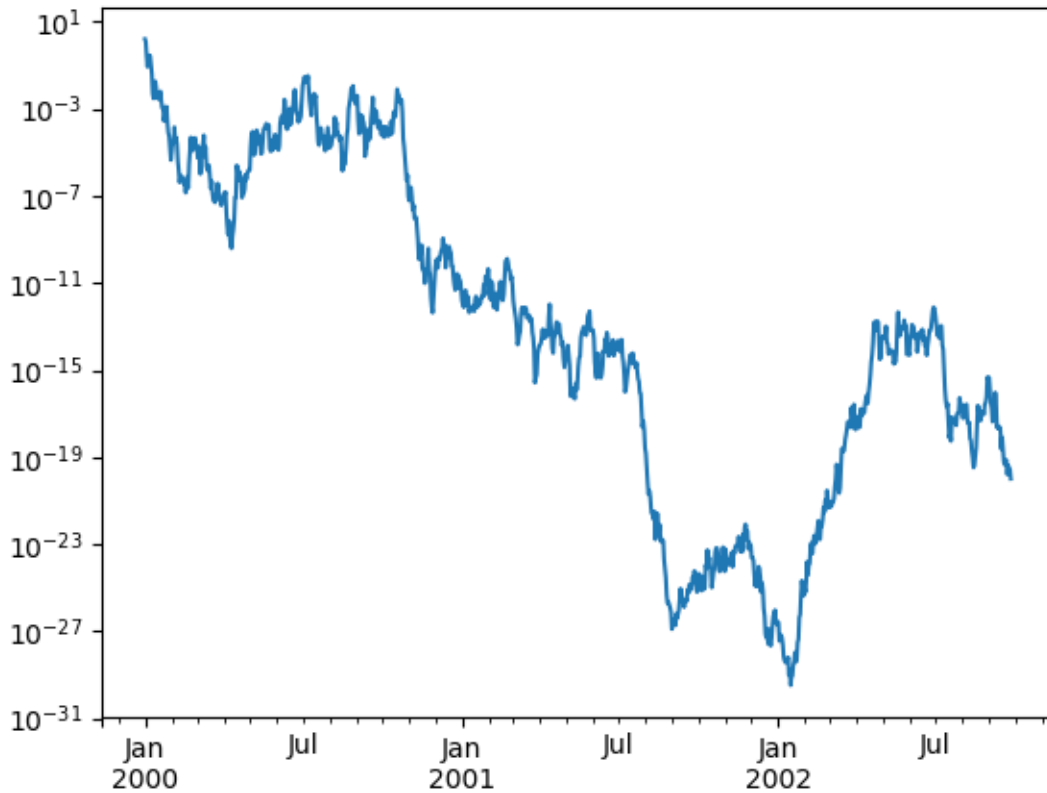


### 22.5.4 Scales

You may pass `logy` to get a log-scale Y axis.

```
In [112]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',  
↳periods=1000))  
  
In [113]: ts = np.exp(ts.cumsum())  
  
In [114]: ts.plot(logy=True)  
Out[114]: <matplotlib.axes._subplots.AxesSubplot at 0x1c2f026080>
```





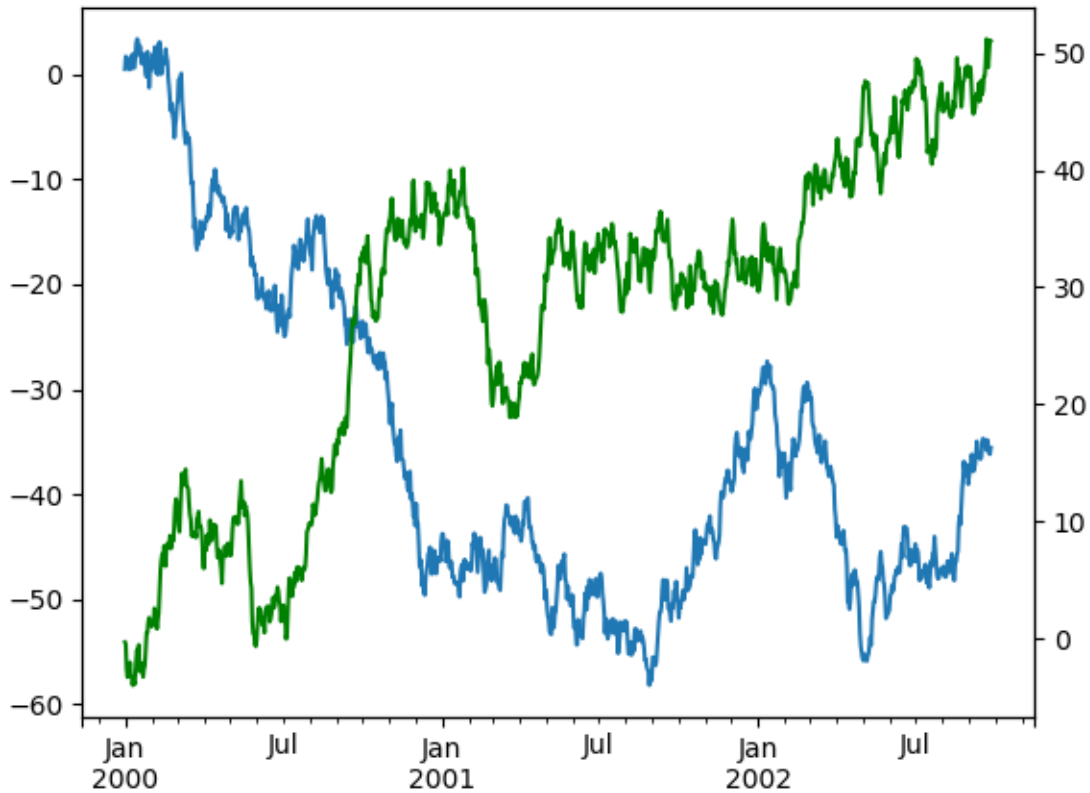
See also the `logx` and `loglog` keyword arguments.

### 22.5.5 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

```
In [115]: df.A.plot()
Out[115]: <matplotlib.axes._subplots.AxesSubplot at 0x1c37e6c7f0>

In [116]: df.B.plot(secondary_y=True, style='g')
Out[116]:
<matplotlib.axes._subplots.AxesSubplot at 0x1c38758ef0>
```



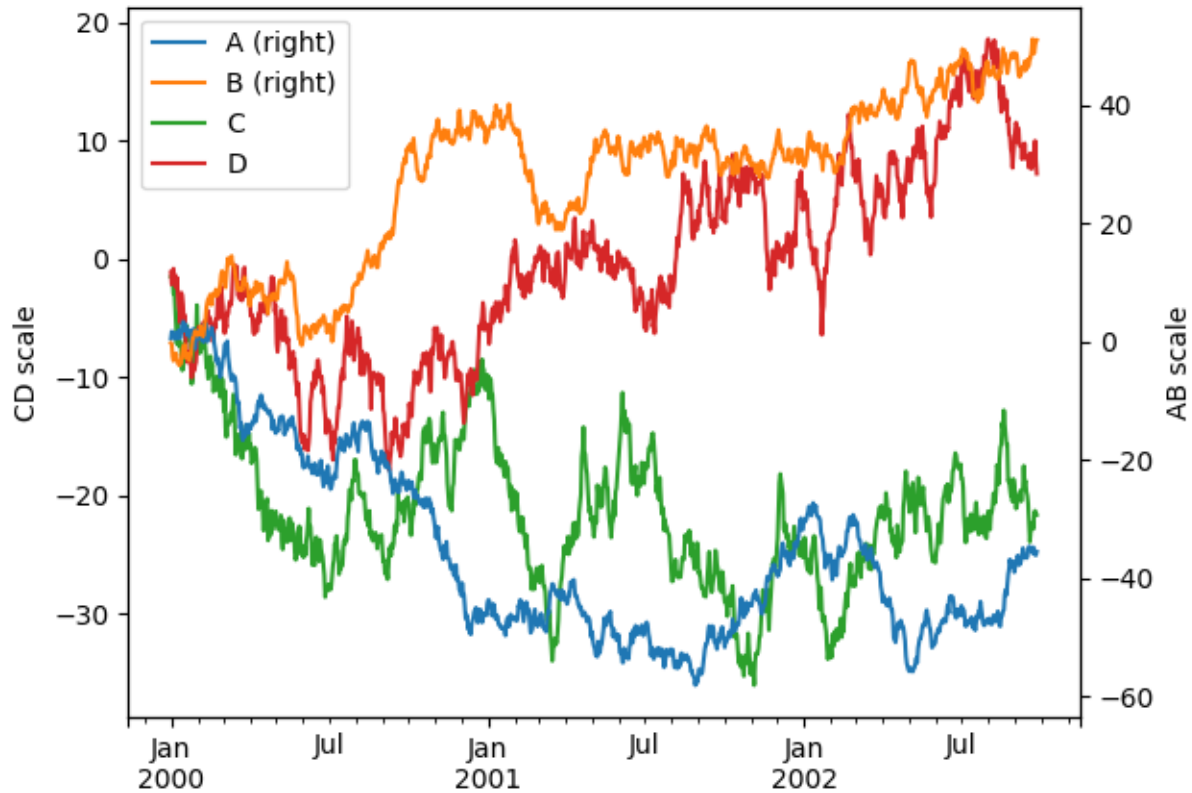
To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```
In [117]: plt.figure()
Out[117]: <Figure size 640x480 with 0 Axes>

In [118]: ax = df.plot(secondary_y=['A', 'B'])

In [119]: ax.set_ylabel('CD scale')
Out[119]: Text(0,0.5,'CD scale')

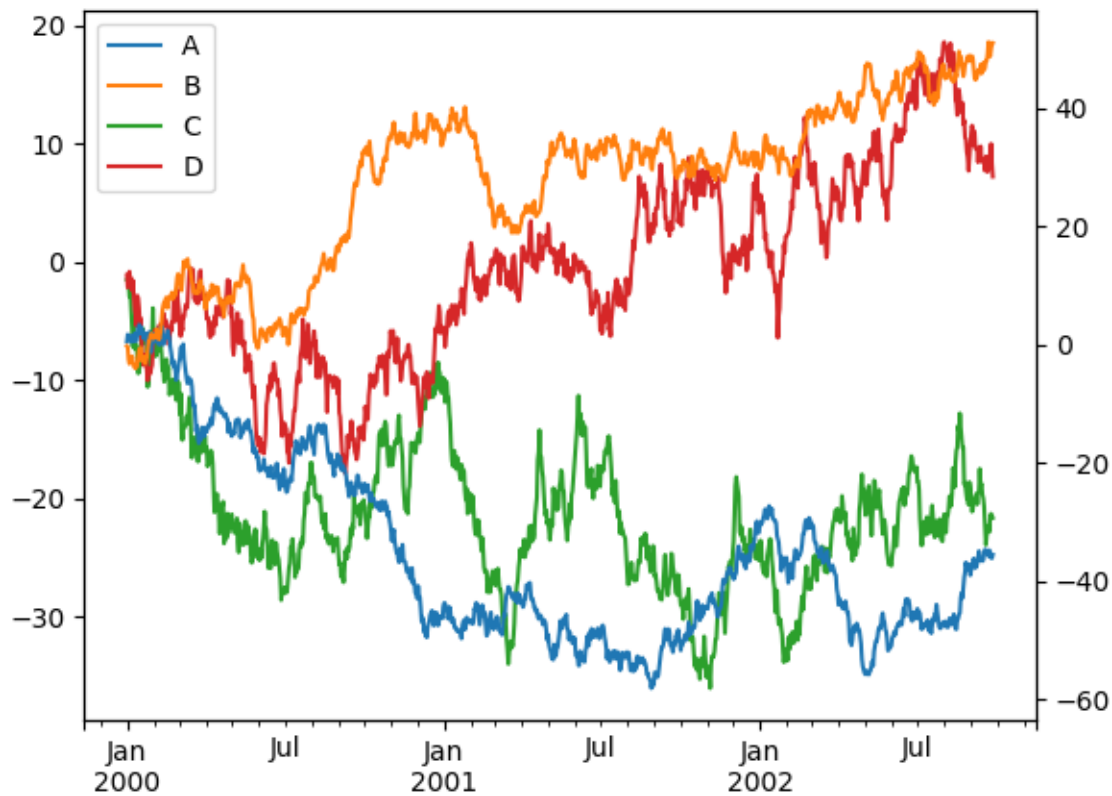
In [120]: ax.right_ax.set_ylabel('AB scale')
Out[120]: Text(0,0.5,'AB scale')
```



Note that the columns plotted on the secondary y-axis is automatically marked with “(right)” in the legend. To turn off the automatic marking, use the `mark_right=False` keyword:

```
In [121]: plt.figure()
Out[121]: <Figure size 640x480 with 0 Axes>

In [122]: df.plot(secondary_y=['A', 'B'], mark_right=False)
Out[122]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c265d6f28>
```



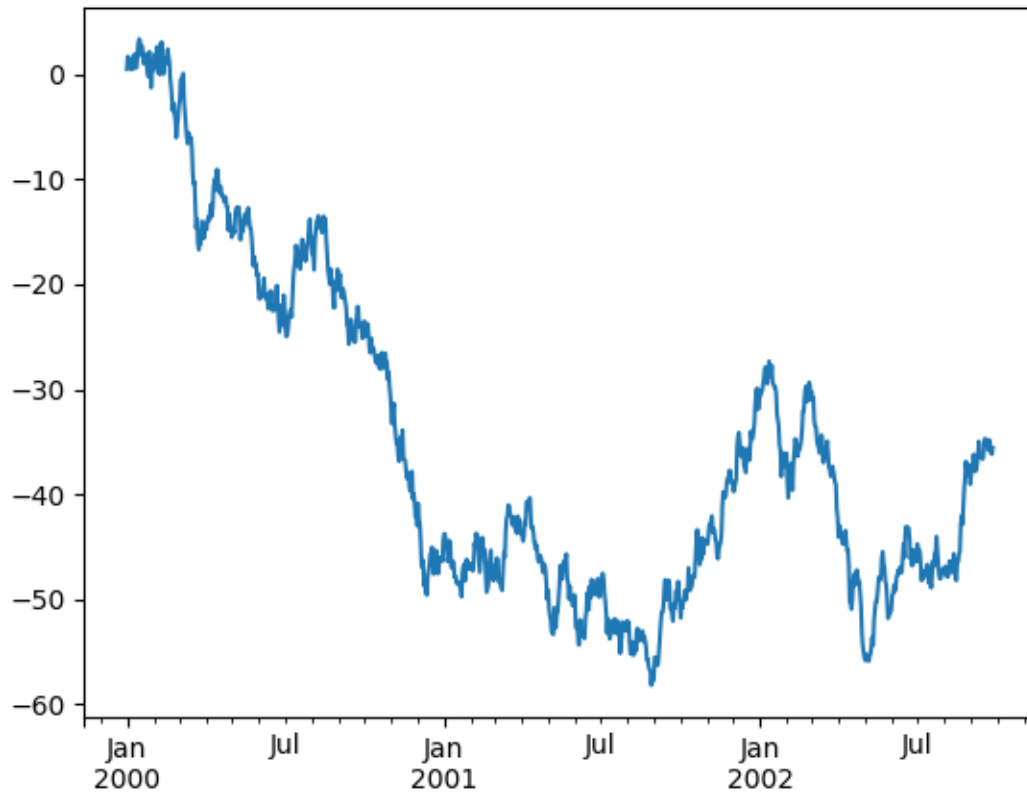
### 22.5.6 Suppressing Tick Resolution Adjustment

pandas includes automatic tick resolution adjustment for regular frequency time-series data. For limited cases where pandas cannot infer the frequency information (e.g., in an externally created `twinx`), you can choose to suppress this behavior for alignment purposes.

Here is the default behavior, notice how the x-axis tick labeling is performed:

```
In [123]: plt.figure()
Out[123]: <Figure size 640x480 with 0 Axes>

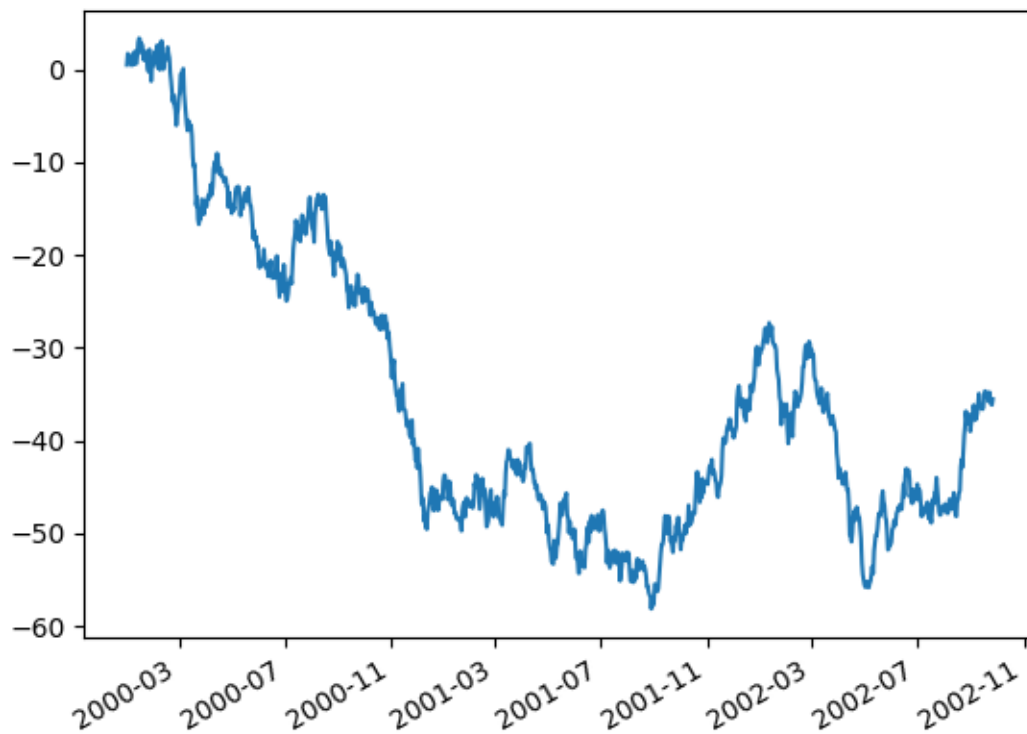
In [124]: df.A.plot()
Out[124]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c31f58a58>
```



Using the `x_compat` parameter, you can suppress this behavior:

```
In [125]: plt.figure()
Out[125]: <Figure size 640x480 with 0 Axes>

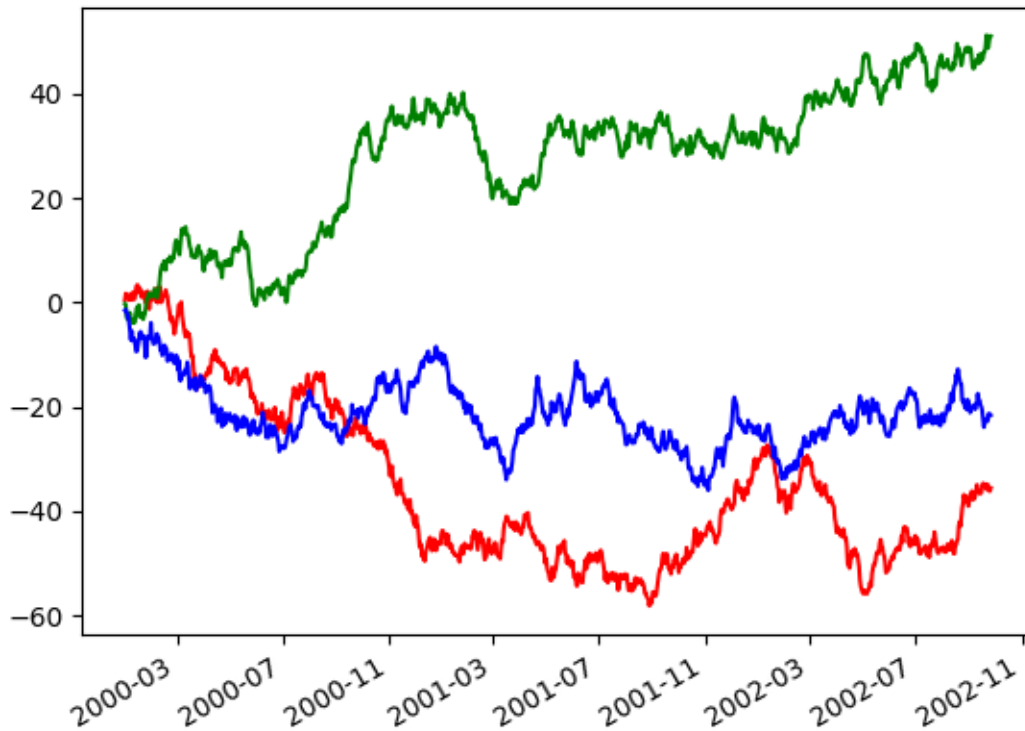
In [126]: df.A.plot(x_compat=True)
Out[126]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c3aba2208>
```



If you have more than one plot that needs to be suppressed, the use method in `pandas.plotting.plot_params` can be used in a *with statement*:

```
In [127]: plt.figure()
Out[127]: <Figure size 640x480 with 0 Axes>

In [128]: with pd.plotting.plot_params.use('x_compat', True):
.....:     df.A.plot(color='r')
.....:     df.B.plot(color='g')
.....:     df.C.plot(color='b')
.....:
```



### 22.5.7 Automatic Date Tick Adjustment

New in version 0.20.0.

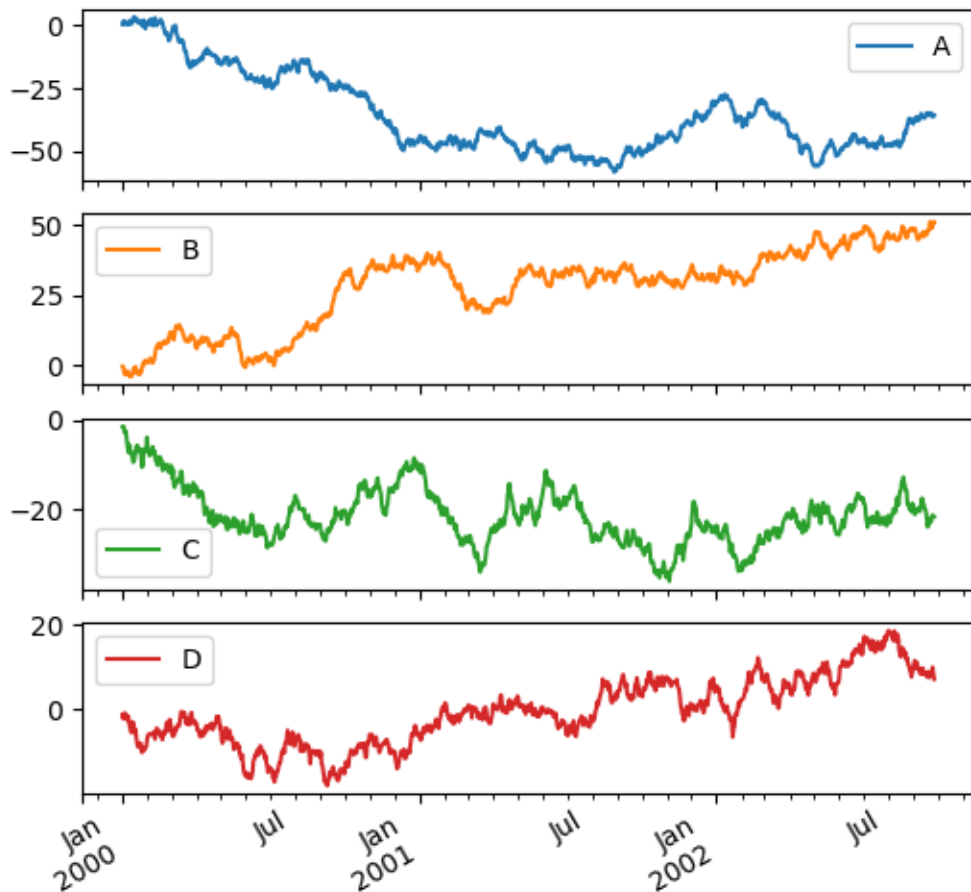
`TimedeltaIndex` now uses the native matplotlib tick locator methods, it is useful to call the automatic date tick adjustment from matplotlib for figures whose ticklabels overlap.

See the `autofmt_xdate` method and the [matplotlib documentation](#) for more.

### 22.5.8 Subplots

Each Series in a DataFrame can be plotted on a different axis with the `subplots` keyword:

```
In [129]: df.plot(subplots=True, figsize=(6, 6));
```



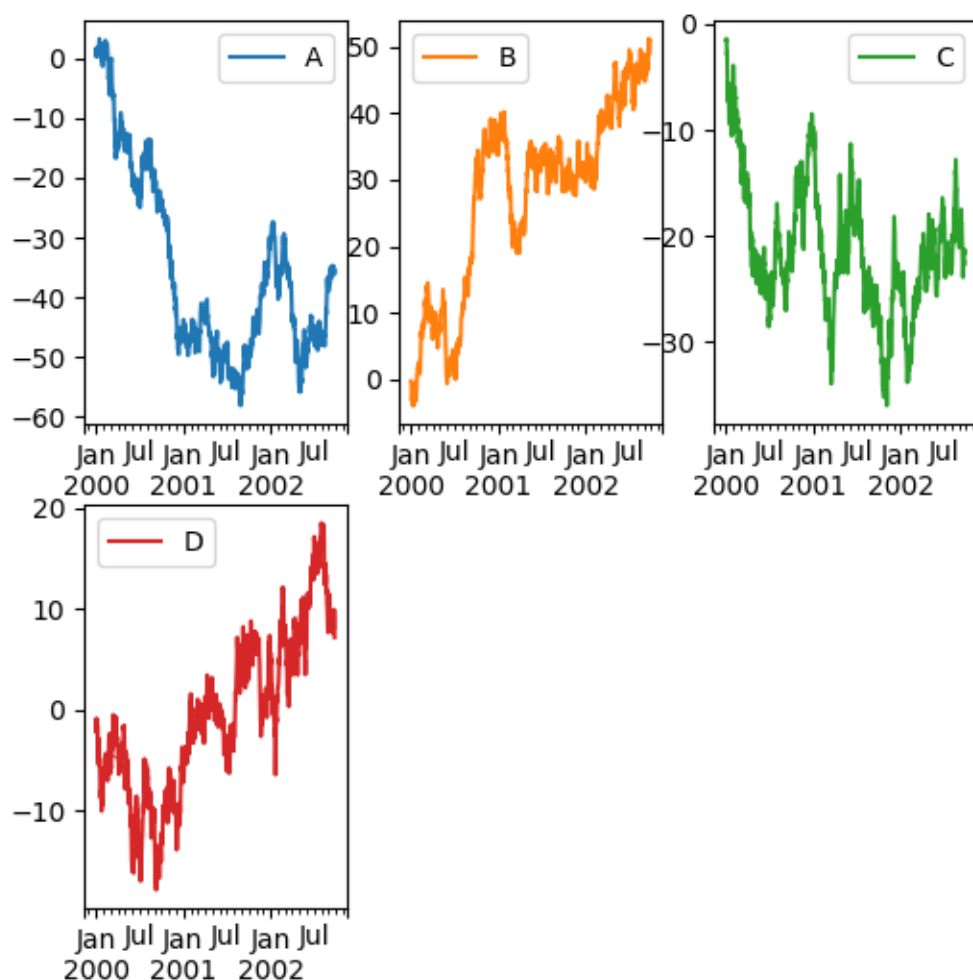
### 22.5.9 Using Layout and Targeting Multiple Axes

The layout of subplots can be specified by the `layout` keyword. It can accept `(rows, columns)`. The `layout` keyword can be used in `hist` and `boxplot` also. If the input is invalid, a `ValueError` will be raised.

The number of axes which can be contained by `rows x columns` specified by `layout` must be larger than the number of required subplots. If `layout` can contain more axes than required, blank axes are not drawn. Similar to a NumPy array's `reshape` method, you can use `-1` for one dimension to automatically calculate the number of rows or columns needed, given the other.

```
In [130]: df.plot(subplots=True, layout=(2, 3), figsize=(6, 6), sharex=False);
```





The above example is identical to using:

```
In [131]: df.plot(subplots=True, layout=(2, -1), figsize=(6, 6), sharex=False);
```

The required number of columns (3) is inferred from the number of series to plot and the given number of rows (2).

You can pass multiple axes created beforehand as list-like via `ax` keyword. This allows more complicated layouts. The passed axes must be the same number as the subplots being drawn.

When multiple axes are passed via the `ax` keyword, `layout`, `sharex` and `sharey` keywords don't affect to the output. You should explicitly pass `sharex=False` and `sharey=False`, otherwise you will see a warning.

```
In [132]: fig, axes = plt.subplots(4, 4, figsize=(6, 6));
```

```
In [133]: plt.subplots_adjust(wspace=0.5, hspace=0.5);
```

```
In [134]: target1 = [axes[0][0], axes[1][1], axes[2][2], axes[3][3]]
```

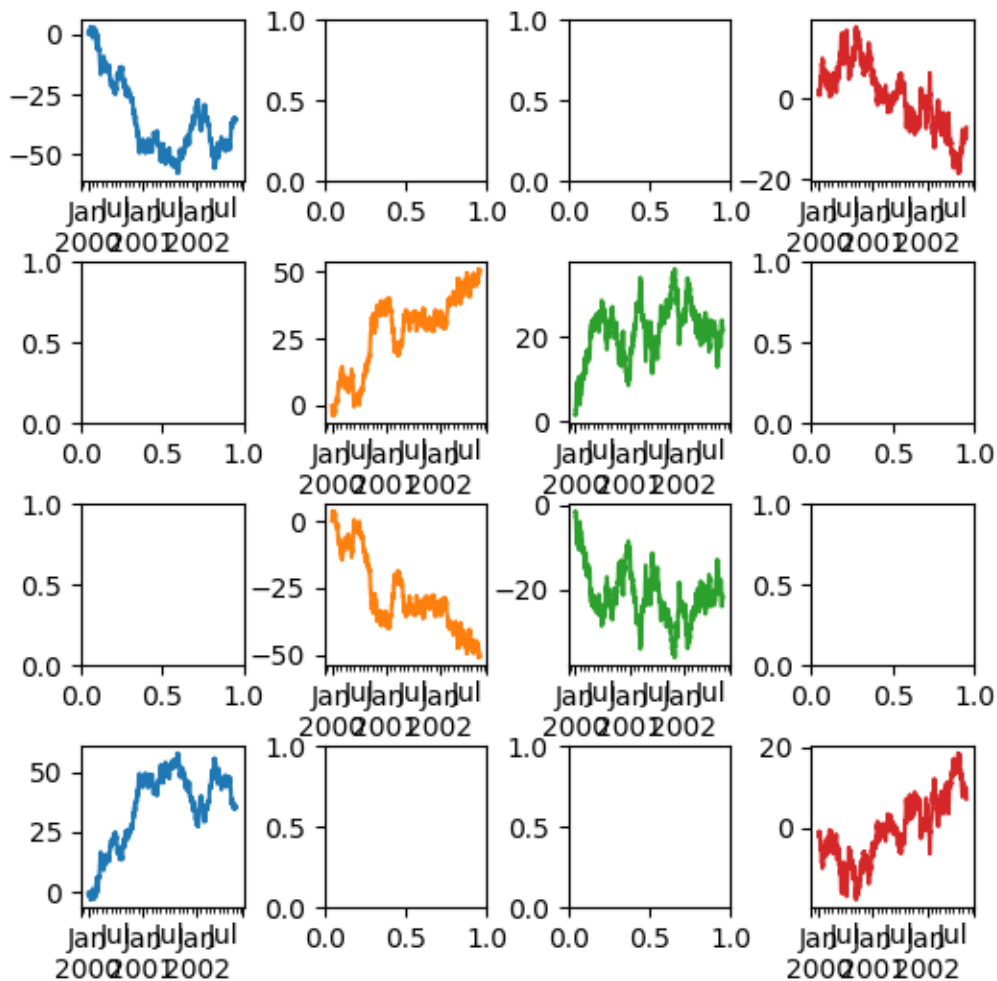
(continues on next page)

(continued from previous page)

```
In [135]: target2 = [axes[3][0], axes[2][1], axes[1][2], axes[0][3]]

In [136]: df.plot(subplots=True, ax=target1, legend=False, sharex=False,
↳sharey=False);

In [137]: (-df).plot(subplots=True, ax=target2, legend=False, sharex=False,
↳sharey=False);
```



Another option is passing an `ax` argument to `Series.plot()` to plot on a particular axis:

```
In [138]: fig, axes = plt.subplots(nrows=2, ncols=2)

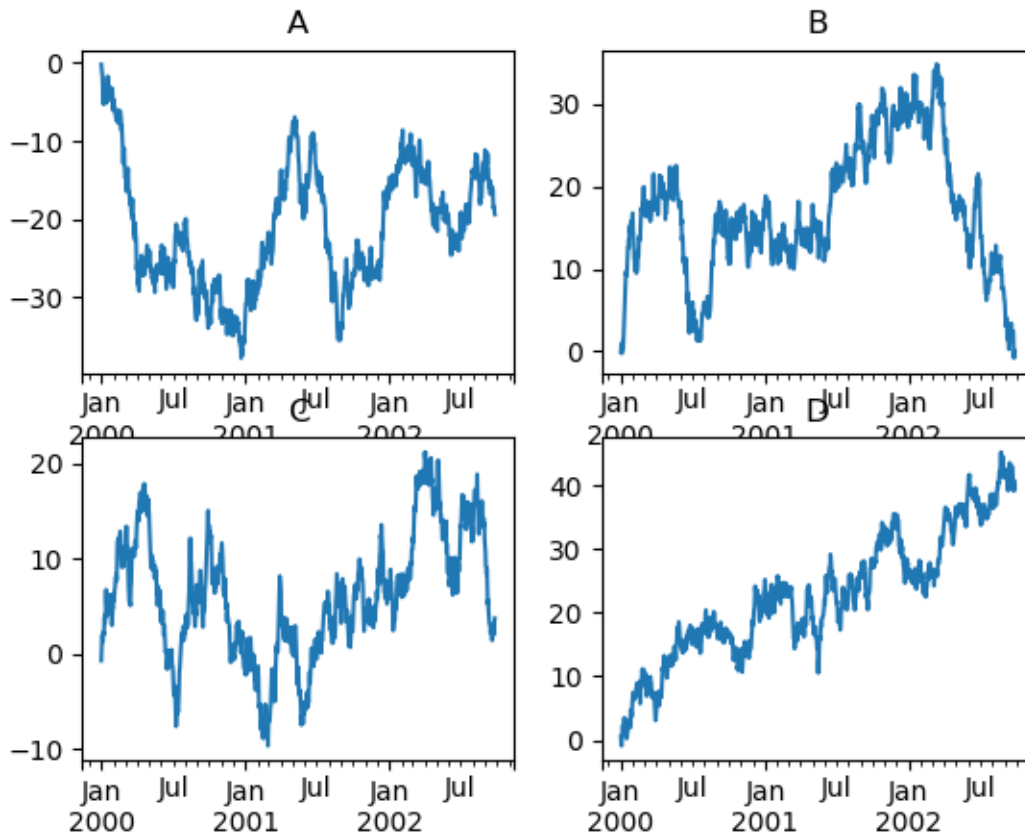
In [139]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A');

In [140]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B');
```

(continues on next page)

(continued from previous page)

```
In [141]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C');
In [142]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D');
```



### 22.5.10 Plotting With Error Bars

Plotting with error bars is supported in `DataFrame.plot()` and `Series.plot()`.

Horizontal and vertical error bars can be supplied to the `xerr` and `yerr` keyword arguments to `plot()`. The error values can be specified using a variety of formats:

- As a `DataFrame` or dict of errors with column names matching the `columns` attribute of the plotting `DataFrame` or matching the `name` attribute of the `Series`.
- As a `str` indicating which of the columns of plotting `DataFrame` contain the error values.
- As raw values (list, tuple, or `np.ndarray`). Must be the same length as the plotting `DataFrame/Series`.

Asymmetrical error bars are also supported, however raw error values must be provided in this case. For a `M` length `Series`, a `Mx2` array should be provided indicating lower and upper (or left and right) errors. For a `MxN` `DataFrame`, asymmetrical errors should be in a `Mx2xN` array.

Here is an example of one way to easily plot group means with standard deviations from the raw data.

```
# Generate the data
In [143]: ix3 = pd.MultiIndex.from_arrays([[ 'a', 'a', 'a', 'a', 'b', 'b', 'b', 'b'], [
↳ 'foo', 'foo', 'bar', 'bar', 'foo', 'foo', 'bar', 'bar']], names=[ 'letter', 'word'])

In [144]: df3 = pd.DataFrame({'data1': [3, 2, 4, 3, 2, 4, 3, 2], 'data2': [6, 5, 7, 5,
↳ 4, 5, 6, 5]}, index=ix3)

# Group by index labels and take the means and standard deviations for each group
In [145]: gp3 = df3.groupby(level=( 'letter', 'word'))

In [146]: means = gp3.mean()

In [147]: errors = gp3.std()

In [148]: means
Out[148]:
```

|        |      | data1 | data2 |
|--------|------|-------|-------|
| letter | word |       |       |
| a      | bar  | 3.5   | 6.0   |
|        | foo  | 2.5   | 5.5   |
| b      | bar  | 2.5   | 5.5   |
|        | foo  | 3.0   | 4.5   |

```

In [149]: errors
\\////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
↳

```

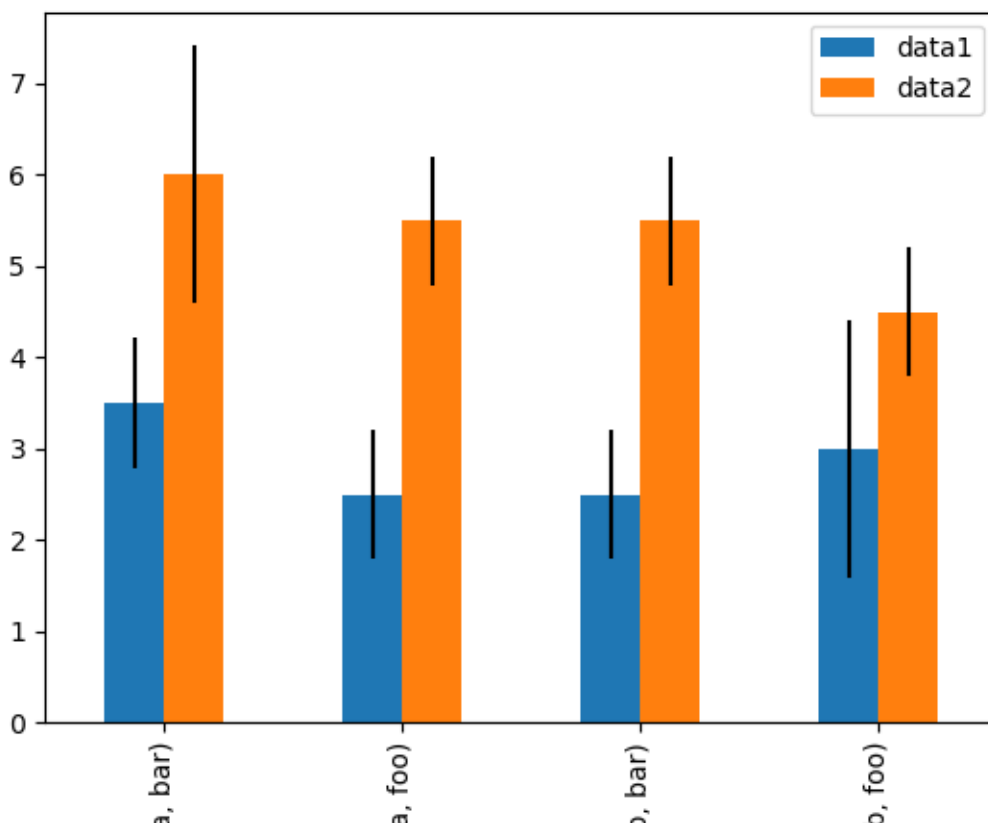
|        |      | data1    | data2    |
|--------|------|----------|----------|
| letter | word |          |          |
| a      | bar  | 0.707107 | 1.414214 |
|        | foo  | 0.707107 | 0.707107 |
| b      | bar  | 0.707107 | 0.707107 |
|        | foo  | 1.414214 | 0.707107 |

```

# Plot
In [150]: fig, ax = plt.subplots()

In [151]: means.plot.bar(yerr=errors, ax=ax)
Out[151]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3f1fd208>

```



### 22.5.11 Plotting Tables

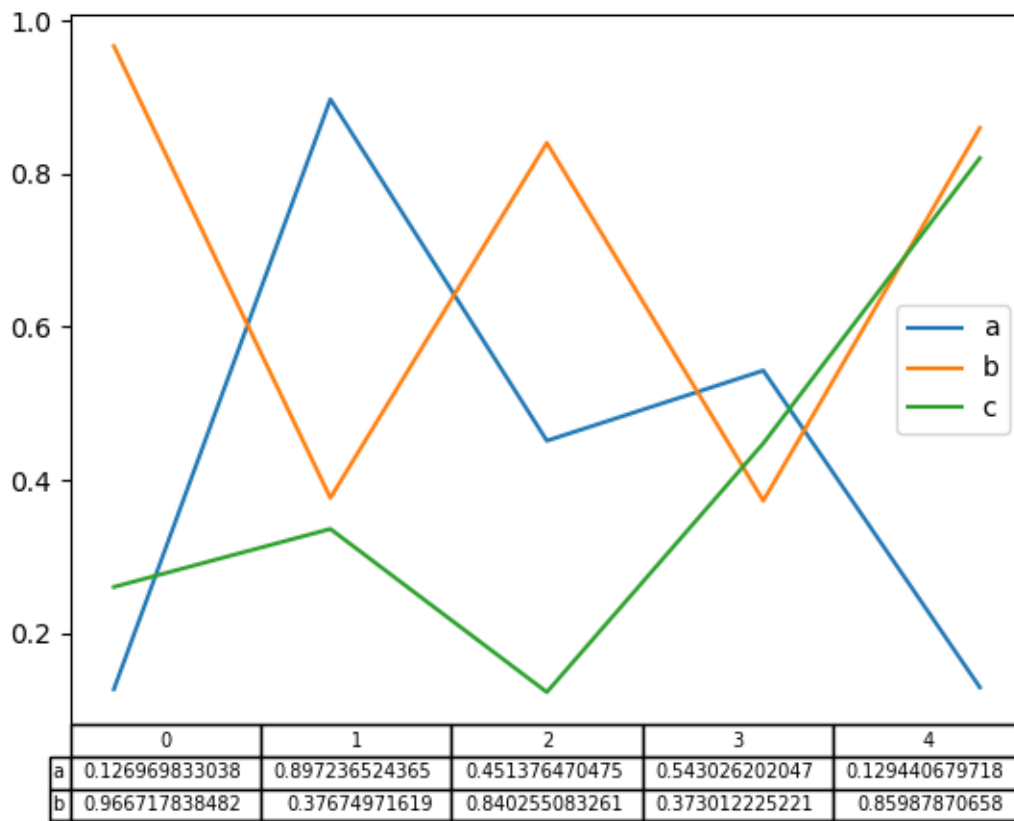
Plotting with matplotlib table is now supported in `DataFrame.plot()` and `Series.plot()` with a `table` keyword. The `table` keyword can accept `bool`, `DataFrame` or `Series`. The simple way to draw a table is to specify `table=True`. Data will be transposed to meet matplotlib's default layout.

```
In [152]: fig, ax = plt.subplots(1, 1)

In [153]: df = pd.DataFrame(np.random.rand(5, 3), columns=['a', 'b', 'c'])

In [154]: ax.get_xaxis().set_visible(False)    # Hide Ticks

In [155]: df.plot(table=True, ax=ax)
Out[155]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3e719550>
```

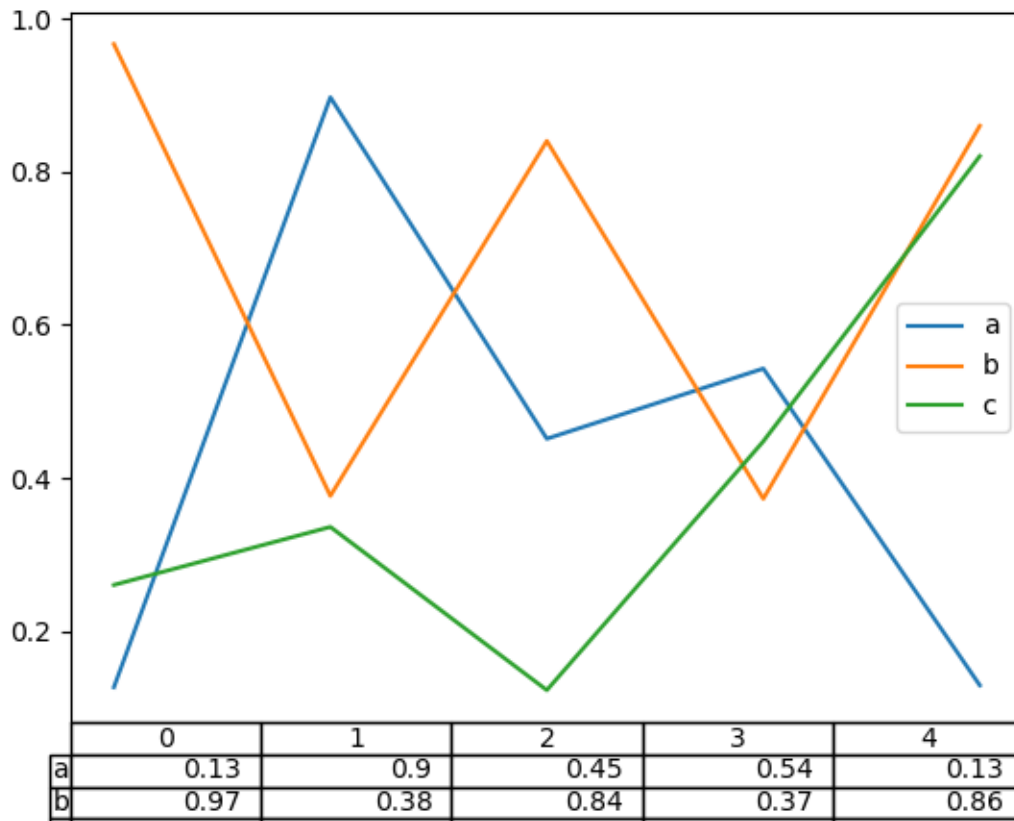


Also, you can pass a different `DataFrame` or `Series` to the `table` keyword. The data will be drawn as displayed in print method (not transposed automatically). If required, it should be transposed manually as seen in the example below.

```
In [156]: fig, ax = plt.subplots(1, 1)

In [157]: ax.get_xaxis().set_visible(False)    # Hide Ticks

In [158]: df.plot(table=np.round(df.T, 2), ax=ax)
Out[158]: <matplotlib.axes._subplots.AxesSubplot at 0x1c3e23f320>
```



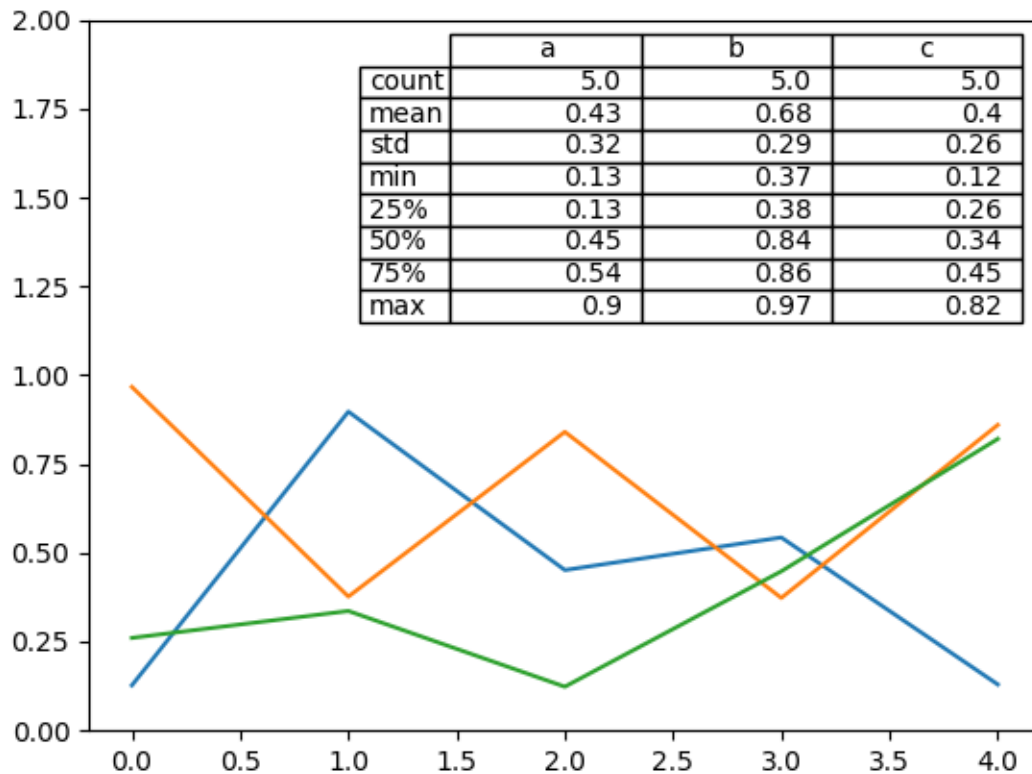
There also exists a helper function `pandas.plotting.table`, which creates a table from *DataFrame* or *Series*, and adds it to an `matplotlib.Axes` instance. This function can accept keywords which the `matplotlib` table has.

```
In [159]: from pandas.plotting import table

In [160]: fig, ax = plt.subplots(1, 1)

In [161]: table(ax, np.round(df.describe(), 2),
.....:         loc='upper right', colWidths=[0.2, 0.2, 0.2])
.....:
Out[161]: <matplotlib.table.Table at 0x1c382b3630>

In [162]: df.plot(ax=ax, ylim=(0, 2), legend=None)
Out[162]: <matplotlib.axes._
↳subplots.AxesSubplot at 0x1c382b48d0>
```



**Note:** You can get table instances on the axes using `axes.tables` property for further decorations. See the [matplotlib table documentation](#) for more.

## 22.5.12 Colormaps

A potential issue when plotting a large number of columns is that it can be difficult to distinguish some series due to repetition in the default colors. To remedy this, `DataFrame` plotting supports the use of the `colormap` argument, which accepts either a Matplotlib [colormap](#) or a string that is a name of a colormap registered with Matplotlib. A visualization of the default matplotlib colormaps is available [here](#).

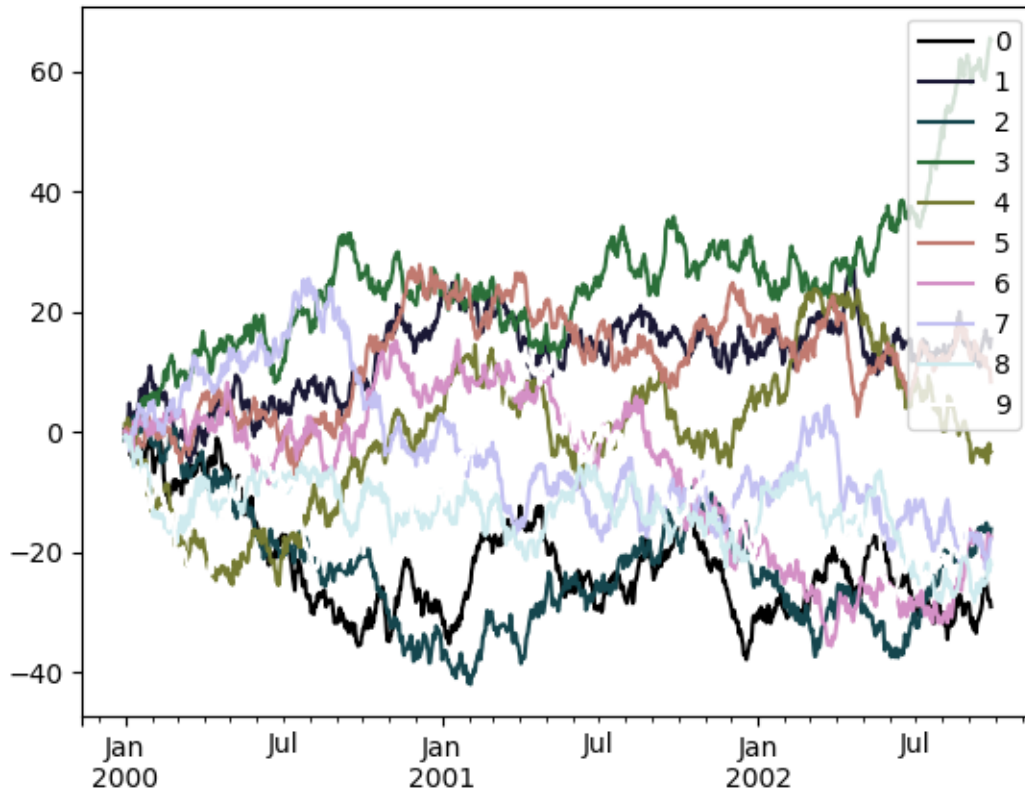
As matplotlib does not directly support colormaps for line-based plots, the colors are selected based on an even spacing determined by the number of columns in the `DataFrame`. There is no consideration made for background color, so some colormaps will produce lines that are not easily visible.

To use the `cubehelix` colormap, we can pass `colormap='cubehelix'`.

```
In [163]: df = pd.DataFrame(np.random.randn(1000, 10), index=ts.index)
In [164]: df = df.cumsum()
In [165]: plt.figure()
Out[165]: <Figure size 640x480 with 0 Axes>
In [166]: df.plot(colormap='cubehelix')
Out[166]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c38057208> (continues on next page)
```



(continued from previous page)

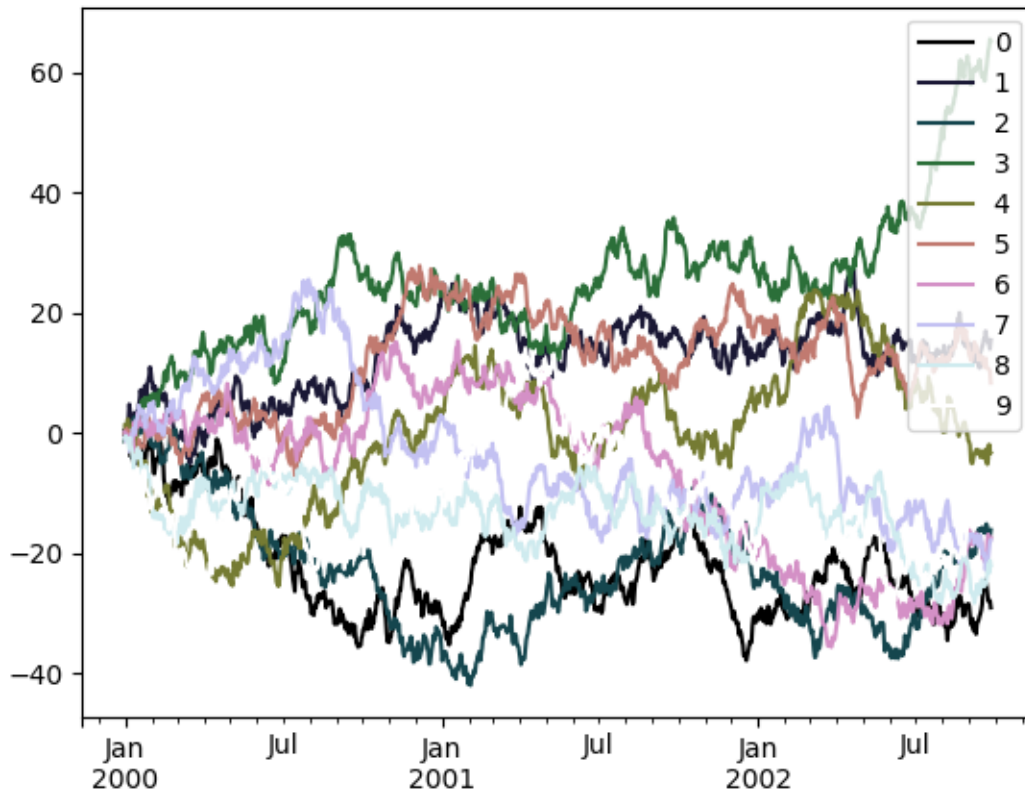


Alternatively, we can pass the colormap itself:

```
In [167]: from matplotlib import cm

In [168]: plt.figure()
Out[168]: <Figure size 640x480 with 0 Axes>

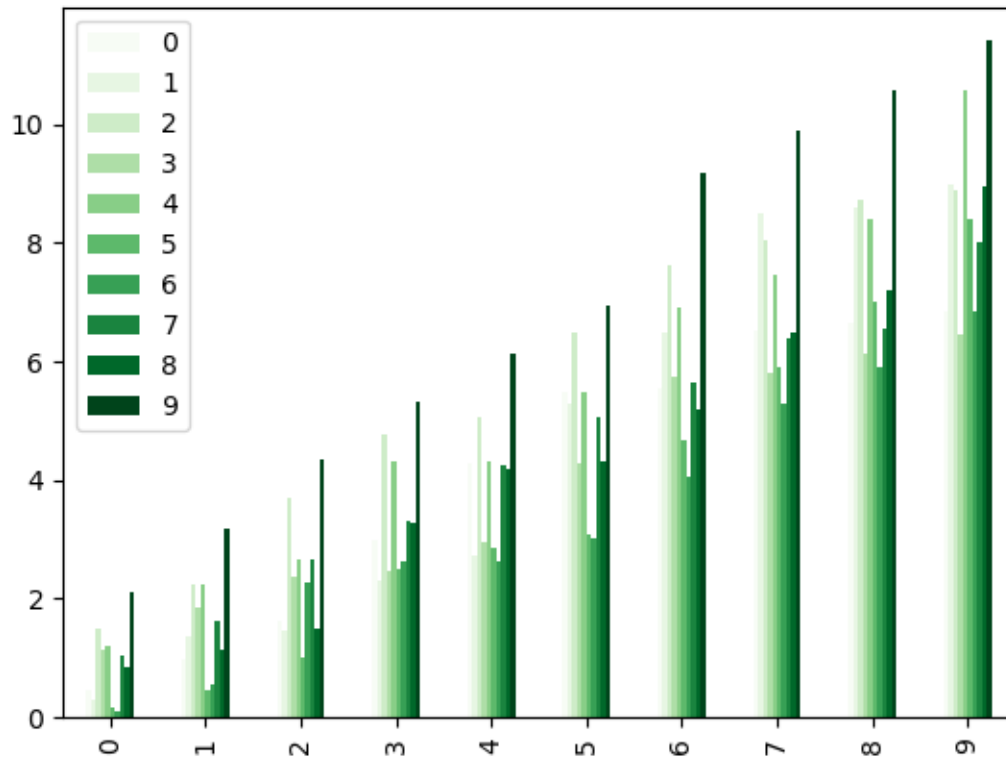
In [169]: df.plot(colormap=cm.cubehelix)
Out[169]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c3e0b20f0>
```



Colormaps can also be used other plot types, like bar charts:

```
In [170]: dd = pd.DataFrame(np.random.randn(10, 10)).applymap(abs)
In [171]: dd = dd.cumsum()
In [172]: plt.figure()
Out[172]: <Figure size 640x480 with 0 Axes>

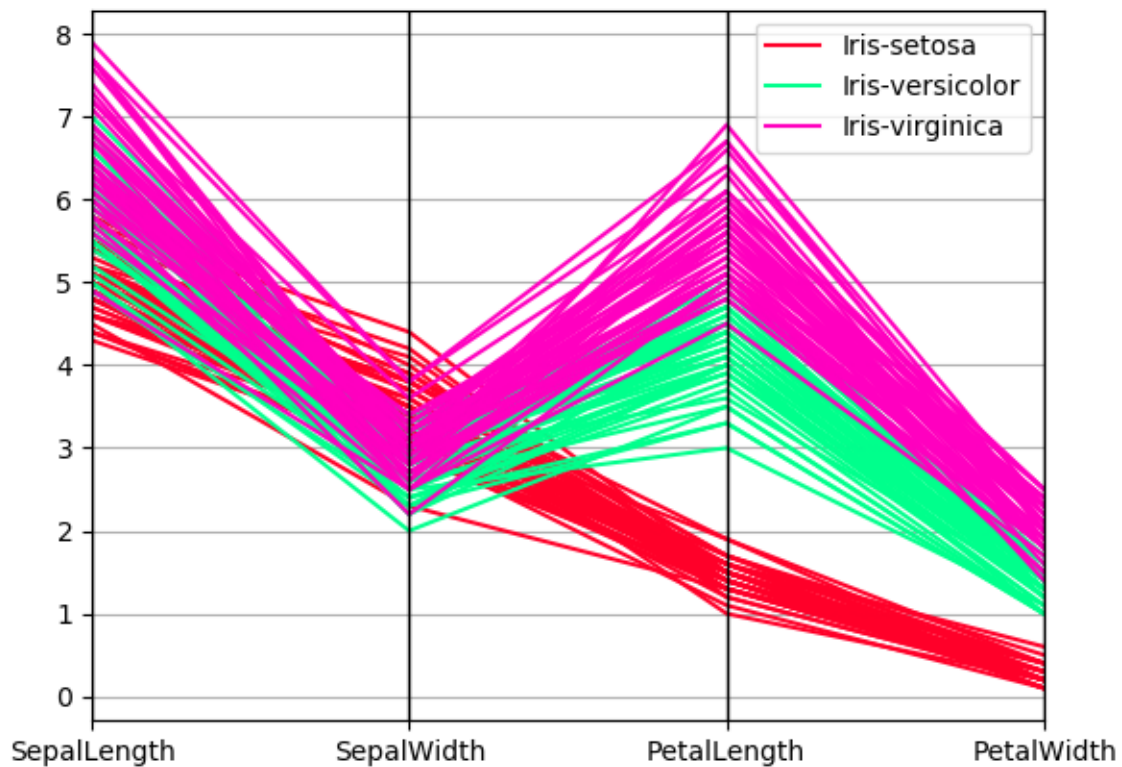
In [173]: dd.plot.bar(colormap='Greens')
Out[173]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c3e7c6b38>
```



Parallel coordinates charts:

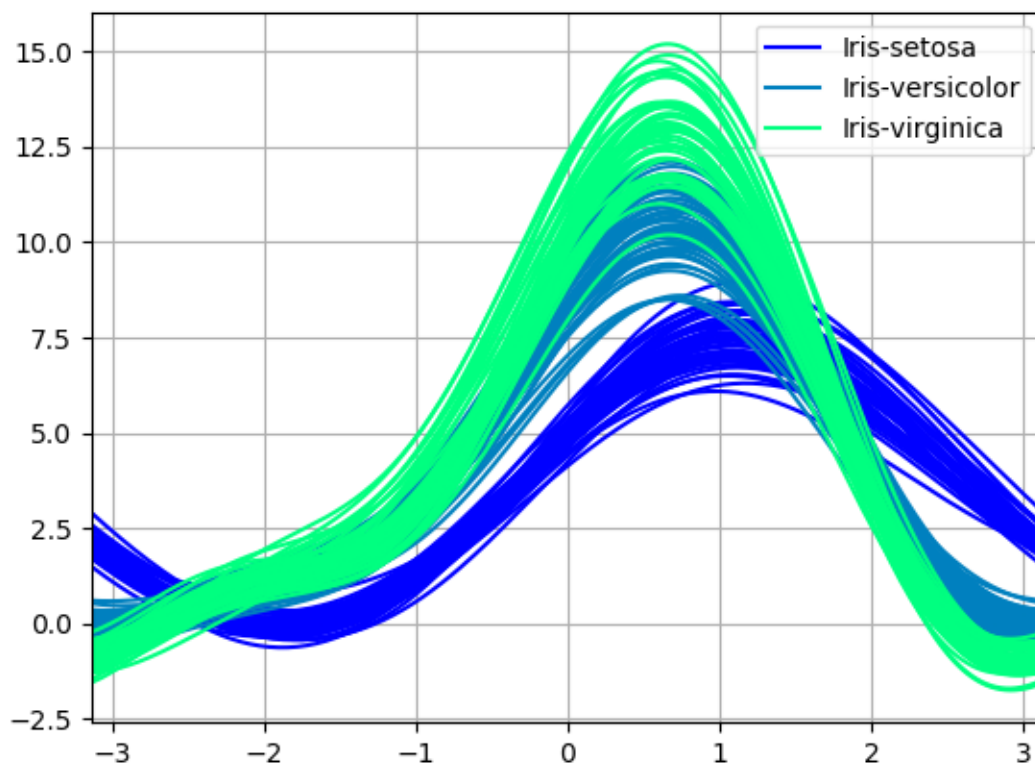
```
In [174]: plt.figure()
Out[174]: <Figure size 640x480 with 0 Axes>

In [175]: parallel_coordinates(data, 'Name', colormap='gist_rainbow')
Out[175]: <matplotlib.axes._subplots.
AxesSubplot at 0x1c3e3f3240>
```



```
In [176]: plt.figure()
Out[176]: <Figure size 640x480 with 0 Axes>

In [177]: andrews_curves(data, 'Name', colormap='winter')
Out[177]: <matplotlib.axes._subplots.
↳ AxesSubplot at 0x1c3ded0080>
```



## 22.6 Plotting directly with matplotlib

In some situations it may still be preferable or necessary to prepare plots directly with matplotlib, for instance when a certain type of plot or customization is not (yet) supported by pandas. `Series` and `DataFrame` objects behave like arrays and can therefore be passed directly to matplotlib functions without explicit casts.

pandas also automatically registers formatters and locators that recognize date indices, thereby extending date and time support to practically all plot types available in matplotlib. Although this formatting does not provide the same level of refinement you would get when plotting via pandas, it can be faster when plotting a large number of points.

```
In [178]: price = pd.Series(np.random.randn(150).cumsum(),
.....:                      index=pd.date_range('2000-1-1', periods=150, freq='B'))
.....:

In [179]: ma = price.rolling(20).mean()

In [180]: mstd = price.rolling(20).std()

In [181]: plt.figure()
Out[181]: <Figure size 640x480 with 0 Axes>

In [182]: plt.plot(price.index, price, 'k')
Out[182]: [<matplotlib.lines.Line2D at 0x1c3fc4f710>]
```

(continues on next page)

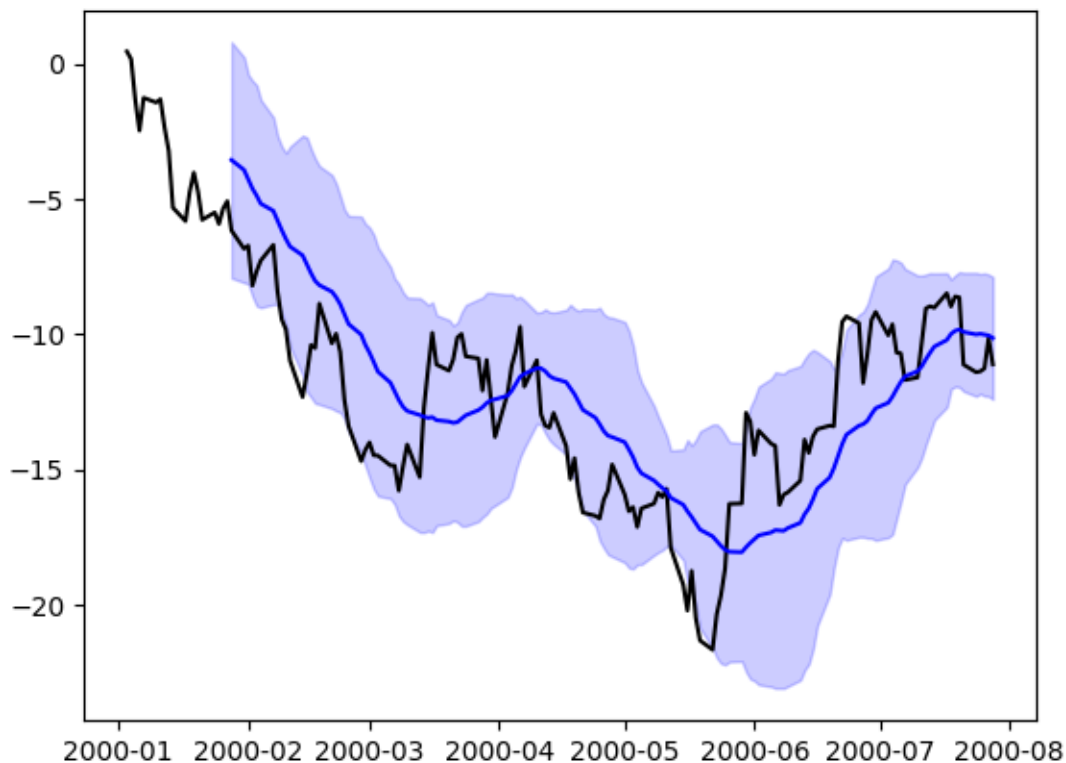
(continued from previous page)

```
In [183]: plt.plot(ma.index, ma, 'b')
```

```
\\Out[183]: <matplotlib.lines.Line2D at 0x1c3fc6aef0>
```

```
In [184]: plt.fill_between(mstd.index, ma-2*mstd, ma+2*mstd, color='b', alpha=0.2)
```

```
\\Out[184]: <matplotlib.collections.PolyCollection at 0x1c304d6a58>
```



## 22.7 Trellis plotting interface

**Warning:** The `rplot` trellis plotting interface has been **removed**. Please use external packages like [seaborn](#) for similar but more refined functionality and refer to our 0.18.1 documentation [here](#) for how to convert to using it.

## STYLING

*New in version 0.17.1*

*Provisional: This is a new feature and still under development. We'll be adding features and possibly making breaking changes in future releases. We'd love to hear your feedback.*

This document is written as a Jupyter Notebook, and can be viewed or downloaded [here](#).

You can apply **conditional formatting**, the visual styling of a DataFrame depending on the data within, by using the `DataFrame.style` property. This is a property that returns a `Styler` object, which has useful methods for formatting and displaying DataFrames.

The styling is accomplished using CSS. You write “style functions” that take scalars, DataFrames or Series, and return *like-indexed* DataFrames or Series with CSS “attribute: value” pairs for the values. These functions can be incrementally passed to the `Styler` which collects the styles before rendering.

### 23.1 Building Styles

Pass your style functions into one of the following methods:

- `Styler.applymap`: elementwise
- `Styler.apply`: column-/row-/table-wise

Both of those methods take a function (and some other keyword arguments) and applies your function to the DataFrame in a certain way. `Styler.applymap` works through the DataFrame elementwise. `Styler.apply` passes each column or row into your DataFrame one-at-a-time or the entire table at once, depending on the `axis` keyword argument. For columnwise use `axis=0`, rowwise use `axis=1`, and for the entire table at once use `axis=None`.

For `Styler.applymap` your function should take a scalar and return a single string with the CSS attribute-value pair.

For `Styler.apply` your function should take a Series or DataFrame (depending on the `axis` parameter), and return a Series or DataFrame with an identical shape where each value is a string with a CSS attribute-value pair.

Let's see some examples.

```
In [2]: import pandas as pd
import numpy as np

np.random.seed(24)
df = pd.DataFrame({'A': np.linspace(1, 10, 10)})
df = pd.concat([df, pd.DataFrame(np.random.randn(10, 4), columns=list('BCDE'))],
               axis=1)
df.iloc[0, 2] = np.nan
```

Here's a boring example of rendering a DataFrame, without any (visible) styles:

```
In [3]: df.style
```

```
Out[3]: <pandas.io.formats.style.Styler at 0x1119c4f98>
```

*Note:* The `DataFrame.style` attribute is a property that returns a `Styler` object. `Styler` has a `_repr_html_` method defined on it so they are rendered automatically. If you want the actual HTML back for further processing or for writing to file call the `.render()` method which returns a string.

The above output looks very similar to the standard `DataFrame` HTML representation. But we've done some work behind the scenes to attach CSS classes to each cell. We can view these by calling the `.render` method.

```
In [4]: df.style.highlight_null().render().split('\n')[10]
```

```
Out[4]: ['<style  type="text/css" >',
        '#T_2baa851c_80a0_11e8_ae45_186590cd1c87row0_col2 {',
        '    background-color:  red;',
        '}</style> ',
        '<table id="T_2baa851c_80a0_11e8_ae45_186590cd1c87" > ',
        '<thead>    <tr> ',
        '        <th class="blank level0" ></th> ',
        '        <th class="col_heading level0 col0" >A</th> ',
        '        <th class="col_heading level0 col1" >B</th> ',
        '        <th class="col_heading level0 col2" >C</th> ']
```

The `row0_col2` is the identifier for that particular cell. We've also prepended each row/column identifier with a UUID unique to each `DataFrame` so that the style from one doesn't collide with the styling from another within the same notebook or page (you can set the `uuid` if you'd like to tie together the styling of two `DataFrames`).

When writing style functions, you take care of producing the CSS attribute / value pairs you want. Pandas matches those up with the CSS classes that identify each cell.

Let's write a simple style function that will color negative numbers red and positive numbers black.

```
In [5]: def color_negative_red(val):
        """
        Takes a scalar and returns a string with
        the css property `color: red` for negative
        strings, black otherwise.
        """
        color = 'red' if val < 0 else 'black'
        return 'color: %s' % color
```

In this case, the cell's style depends only on it's own value. That means we should use the `Styler.applymap` method which works elementwise.

```
In [6]: s = df.style.applymap(color_negative_red)
        s
```

```
Out[6]: <pandas.io.formats.style.Styler at 0x113d34048>
```

Notice the similarity with the standard `df.applymap`, which operates on `DataFrames` elementwise. We want you to be able to reuse your existing knowledge of how to interact with `DataFrames`.

Notice also that our function returned a string containing the CSS attribute and value, separated by a colon just like in a `<style>` tag. This will be a common theme.

Finally, the input shapes matched. `Styler.applymap` calls the function on each scalar input, and the function returns a scalar output.

Now suppose you wanted to highlight the maximum value in each column. We can't use `.applymap` anymore since that operated elementwise. Instead, we'll turn to `.apply` which operates columnwise (or rowwise using the `axis` keyword). Later on we'll see that something like `highlight_max` is already defined on `Styler` so you wouldn't need to write this yourself.



```
In [7]: def highlight_max(s):
        '''
        highlight the maximum in a Series yellow.
        '''
        is_max = s == s.max()
        return ['background-color: yellow' if v else '' for v in is_max]
```

```
In [8]: df.style.apply(highlight_max)
```

```
Out[8]: <pandas.io.formats.style.Styler at 0x113cec9e8>
```

In this case the input is a Series, one column at a time. Notice that the output shape of `highlight_max` matches the input shape, an array with `len(s)` items.

We encourage you to use method chains to build up a style piecewise, before finally rendering at the end of the chain.

```
In [9]: df.style.\
        applymap(color_negative_red).\
        apply(highlight_max)
```

```
Out[9]: <pandas.io.formats.style.Styler at 0x113cec160>
```

Above we used `Styler.apply` to pass in each column one at a time.

*Debugging Tip:* If you're having trouble writing your style function, try just passing it into `DataFrame.apply`. Internally, `Styler.apply` uses `DataFrame.apply` so the result should be the same.

What if you wanted to highlight just the maximum value in the entire table? Use `.apply(function, axis=None)` to indicate that your function wants the entire table, not one column or row at a time. Let's try that next.

We'll rewrite our `highlight_max` to handle either Series (from `.apply(axis=0 or 1)`) or DataFrames (from `.apply(axis=None)`). We'll also allow the color to be adjustable, to demonstrate that `.apply`, and `.applymap` pass along keyword arguments.

```
In [10]: def highlight_max(data, color='yellow'):
        '''
        highlight the maximum in a Series or DataFrame
        '''
        attr = 'background-color: {}'.format(color)
        if data.ndim == 1: # Series from .apply(axis=0) or axis=1
            is_max = data == data.max()
            return [attr if v else '' for v in is_max]
        else: # from .apply(axis=None)
            is_max = data == data.max().max()
            return pd.DataFrame(np.where(is_max, attr, ''),
                               index=data.index, columns=data.columns)
```

When using `Styler.apply(func, axis=None)`, the function must return a DataFrame with the same index and column labels.

```
In [11]: df.style.apply(highlight_max, color='darkorange', axis=None)
```

```
Out[11]: <pandas.io.formats.style.Styler at 0x113d214e0>
```

### 23.1.1 Building Styles Summary

Style functions should return strings with one or more CSS attribute: value delimited by semicolons. Use

- `Styler.applymap(func)` for elementwise styles
- `Styler.apply(func, axis=0)` for columnwise styles
- `Styler.apply(func, axis=1)` for rowwise styles

- `Styler.apply(func, axis=None)` for tablewise styles

And crucially the input and output shapes of `func` must match. If `x` is the input then `func(x).shape == x.shape`.

## 23.2 Finer Control: Slicing

Both `Styler.apply`, and `Styler.applymap` accept a `subset` keyword. This allows you to apply styles to specific rows or columns, without having to code that logic into your `style` function.

The value passed to `subset` behaves similar to slicing a `DataFrame`.

- A scalar is treated as a column label
- A list (or series or numpy array)
- A tuple is treated as `(row_indexer, column_indexer)`

Consider using `pd.IndexSlice` to construct the tuple for the last one.

```
In [12]: df.style.apply(highlight_max, subset=['B', 'C', 'D'])
```

```
Out[12]: <pandas.io.formats.style.Styler at 0x113d217b8>
```

For row and column slicing, any valid indexer to `.loc` will work.

```
In [13]: df.style.applymap(color_negative_red,
                          subset=pd.IndexSlice[2:5, ['B', 'D']])
```

```
Out[13]: <pandas.io.formats.style.Styler at 0x113d21a20>
```

Only label-based slicing is supported right now, not positional.

If your style function uses a `subset` or `axis` keyword argument, consider wrapping your function in a `functools.partial`, partialing out that keyword.

```
my_func2 = functools.partial(my_func, subset=42)
```

## 23.3 Finer Control: Display Values

We distinguish the *display* value from the *actual* value in `Styler`. To control the display value, the text is printed in each cell, use `Styler.format`. Cells can be formatted according to a *format spec string* or a callable that takes a single value and returns a string.

```
In [14]: df.style.format("{:.2%}")
```

```
Out[14]: <pandas.io.formats.style.Styler at 0x113d21dd8>
```

Use a dictionary to format specific columns.

```
In [15]: df.style.format({'B': "{:0<4.0f}", 'D': '{:+.2f}'})
```

```
Out[15]: <pandas.io.formats.style.Styler at 0x113d214a8>
```

Or pass in a callable (or dictionary of callables) for more flexible handling.

```
In [16]: df.style.format({"B": lambda x: "±{:.2f}".format(abs(x))})
```

```
Out[16]: <pandas.io.formats.style.Styler at 0x113d21b38>
```

## 23.4 Builtin Styles

Finally, we expect certain styling functions to be common enough that we’ve included a few “built-in” to the `Styler`, so you don’t have to write them yourself.

```
In [17]: df.style.highlight_null(null_color='red')
Out[17]: <pandas.io.formats.style.Styler at 0x113d216d8>
```

You can create “heatmaps” with the `background_gradient` method. These require `matplotlib`, and we’ll use `Seaborn` to get a nice colormap.

```
In [18]: import seaborn as sns

        cm = sns.light_palette("green", as_cmap=True)

        s = df.style.background_gradient(cmap=cm)
        s

/Users/taugspurger/miniconda3/envs/travis-36-doc/lib/python3.6/site-packages/matplotlib/colors.py:504:
  xa[xa < 0] = -1
```

```
Out[18]: <pandas.io.formats.style.Styler at 0x113d21198>
```

`Styler.background_gradient` takes the keyword arguments `low` and `high`. Roughly speaking these extend the range of your data by `low` and `high` percent so that when we convert the colors, the colormap’s entire range isn’t used. This is useful so that you can actually read the text still.

```
In [19]: # Uses the full color range
        df.loc[:4].style.background_gradient(cmap='viridis')

/Users/taugspurger/miniconda3/envs/travis-36-doc/lib/python3.6/site-packages/matplotlib/colors.py:504:
  xa[xa < 0] = -1
```

```
Out[19]: <pandas.io.formats.style.Styler at 0x113d21a58>
```

```
In [20]: # Compress the color range
        (df.loc[:4]
         .style
         .background_gradient(cmap='viridis', low=.5, high=0)
         .highlight_null('red'))
```

```
/Users/taugspurger/miniconda3/envs/travis-36-doc/lib/python3.6/site-packages/matplotlib/colors.py:504:
  xa[xa < 0] = -1
```

```
Out[20]: <pandas.io.formats.style.Styler at 0x1a15a64940>
```

There’s also `.highlight_min` and `.highlight_max`.

```
In [21]: df.style.highlight_max(axis=0)
Out[21]: <pandas.io.formats.style.Styler at 0x113d215c0>
```

Use `Styler.set_properties` when the style doesn’t actually depend on the values.

```
In [22]: df.style.set_properties(**{'background-color': 'black',
                                   'color': 'lawngreen',
                                   'border-color': 'white'})
```

```
Out[22]: <pandas.io.formats.style.Styler at 0x1a15a64908>
```

### 23.4.1 Bar charts

You can include “bar charts” in your `DataFrame`.

```
In [23]: df.style.bar(subset=['A', 'B'], color='#d65f5f')
```

```
Out[23]: <pandas.io.formats.style.Styler at 0x1a15aea4e0>
```

New in version 0.20.0 is the ability to customize further the bar chart: You can now have the `df.style.bar` be centered on zero or midpoint value (in addition to the already existing way of having the min value at the left side of the cell), and you can pass a list of `[color_negative, color_positive]`.

Here's how you can change the above with the new `align='mid'` option:

```
In [24]: df.style.bar(subset=['A', 'B'], align='mid', color=['#d65f5f', '#5fba7d'])
```

```
Out[24]: <pandas.io.formats.style.Styler at 0x1a15bbbe10>
```

The following example aims to give a highlight of the behavior of the new align options:

```
In [25]: import pandas as pd
         from IPython.display import HTML

         # Test series
         test1 = pd.Series([-100,-60,-30,-20], name='All Negative')
         test2 = pd.Series([10,20,50,100], name='All Positive')
         test3 = pd.Series([-10,-5,0,90], name='Both Pos and Neg')

         head = """
         <table>
             <thead>
                 <th>Align</th>
                 <th>All Negative</th>
                 <th>All Positive</th>
                 <th>Both Neg and Pos</th>
             </thead>
             </tbody>

         """

         aligns = ['left', 'zero', 'mid']
         for align in aligns:
             row = "<tr><th>{}</th>".format(align)
             for serie in [test1, test2, test3]:
                 s = serie.copy()
                 s.name = ' '
                 row += "<td>{}</td>".format(s.to_frame().style.bar(align=align,
                                                                     color=['#d65f5f', '#5fba7d'],
                                                                     width=100).render()) #testn['wid

             row += '</tr>'
             head += row

         head+= """
         </tbody>
         </table>"""

         HTML(head)

Out[25]: <IPython.core.display.HTML object>
```

## 23.5 Sharing Styles

Say you have a lovely style built up for a `DataFrame`, and now you want to apply the same style to a second `DataFrame`. Export the style with `df1.style.export`, and import it on the second `DataFrame` with `df1.style.set`

```
In [26]: df2 = -df
         style1 = df.style.applymap(color_negative_red)
         style1

Out[26]: <pandas.io.formats.style.Styler at 0x1a15aea6d8>

In [27]: style2 = df2.style
         style2.use(style1.export())
         style2

Out[27]: <pandas.io.formats.style.Styler at 0x1a15be6978>
```

Notice that you're able share the styles even though they're data aware. The styles are re-evaluated on the new `DataFrame` they've been used upon.

## 23.6 Other Options

You've seen a few methods for data-driven styling. `Styler` also provides a few other options for styles that don't depend on the data.

- precision
- captions
- table-wide styles
- hiding the index or columns

Each of these can be specified in two ways:

- A keyword argument to `Styler.__init__`
- A call to one of the `.set_` or `.hide_` methods, e.g. `.set_caption` or `.hide_columns`

The best method to use depends on the context. Use the `Styler` constructor when building many styled `DataFrames` that should all share the same properties. For interactive use, the `.set_` and `.hide_` methods are more convenient.

### 23.6.1 Precision

You can control the precision of floats using pandas' regular `display.precision` option.

```
In [28]: with pd.option_context('display.precision', 2):
         html = (df.style
               .applymap(color_negative_red)
               .apply(highlight_max))

         html

Out[28]: <pandas.io.formats.style.Styler at 0x1a15aea518>
```

Or through a `set_precision` method.

```
In [29]: df.style\
         .applymap(color_negative_red)\
         .apply(highlight_max)\
         .set_precision(2)
```

```
Out[29]: <pandas.io.formats.style.Styler at 0x1a15bdf400>
```

Setting the precision only affects the printed number; the full-precision values are always passed to your style functions. You can always use `df.round(2).style` if you'd prefer to round from the start.

## 23.6.2 Captions

Regular table captions can be added in a few ways.

```
In [30]: df.style.set_caption('Colormaps, with a caption.')\
        .background_gradient(cmap=cm)
```

```
/Users/taugspurger/miniconda3/envs/travis-36-doc/lib/python3.6/site-packages/matplotlib/colors.py:50:
  xa[xa < 0] = -1
```

```
Out[30]: <pandas.io.formats.style.Styler at 0x1a15aea3c8>
```

## 23.6.3 Table Styles

The next option you have are “table styles”. These are styles that apply to the table as a whole, but don’t look at the data. Certain sytlings, including pseudo-selectors like `:hover` can only be used this way.

```
In [31]: from IPython.display import HTML
```

```
def hover(hover_color="#ffff99"):
    return dict(selector="tr:hover",
                props=[("background-color", "%s" % hover_color)])

styles = [
    hover(),
    dict(selector="th", props=[("font-size", "150%"),
                                ("text-align", "center")]),
    dict(selector="caption", props=[("caption-side", "bottom")])
]
html = (df.style.set_table_styles(styles)
        .set_caption("Hover to highlight.")).render()
html
```

```
Out[31]: <pandas.io.formats.style.Styler at 0x1a15bdfbe0>
```

`table_styles` should be a list of dictionaries. Each dictionary should have the `selector` and `props` keys. The value for `selector` should be a valid CSS selector. Recall that all the styles are already attached to an `id`, unique to each `Styler`. This selector is in addition to that `id`. The value for `props` should be a list of tuples of ('attribute', 'value').

`table_styles` are extremely flexible, but not as fun to type out by hand. We hope to collect some useful ones either in pandas, or preferable in a new package that *builds on top* the tools here.

## 23.6.4 Hiding the Index or Columns

The index can be hidden from rendering by calling `Styler.hide_index`. Columns can be hidden from rendering by calling `Styler.hide_columns` and passing in the name of a column, or a slice of columns.

```
In [32]: df.style.hide_index()
```

```
Out[32]: <pandas.io.formats.style.Styler at 0x1a15aea4a8>
```

```
In [33]: df.style.hide_columns(['C', 'D'])
```

```
Out[33]: <pandas.io.formats.style.Styler at 0x1a15be6b70>
```

### 23.6.5 CSS Classes

Certain CSS classes are attached to cells.

- Index and Column names include `index_name` and `level<k>` where `k` is its level in a `MultiIndex`
- Index label cells include
  - `row_heading`
  - `row<n>` where `n` is the numeric position of the row
  - `level<k>` where `k` is the level in a `MultiIndex`
- Column label cells include
  - `col_heading`
  - `col<n>` where `n` is the numeric position of the column
  - `level<k>` where `k` is the level in a `MultiIndex`
- Blank cells include `blank`
- Data cells include `data`

### 23.6.6 Limitations

- `DataFrame` only (use `Series.to_frame().style`)
- The index and columns must be unique
- No large repr, and performance isn't great; this is intended for summary `DataFrames`
- You can only style the *values*, not the index or columns
- You can only apply styles, you can't insert new HTML entities

Some of these will be addressed in the future.

### 23.6.7 Terms

- **Style function:** a function that's passed into `Styler.apply` or `Styler.applymap` and returns values like `'css attribute: value'`
- **Builtin style functions:** style functions that are methods on `Styler`
- **table style:** a dictionary with the two keys `selector` and `props`. `selector` is the CSS selector that `props` will apply to. `props` is a list of `(attribute, value)` tuples. A list of table styles passed into `Styler`.

## 23.7 Fun stuff

Here are a few interesting examples.

`Styler` interacts pretty well with widgets. If you're viewing this online instead of running the notebook yourself, you're missing out on interactively adjusting the color palette.

```
In [34]: from IPython.html import widgets
         @widgets.interact
         def f(h_neg=(0, 359, 1), h_pos=(0, 359), s=(0., 99.9), l=(0., 99.9)):
             return df.style.background_gradient(
                 cmap=sns.palettes.diverging_palette(h_neg=h_neg, h_pos=h_pos, s=s, l=l,
                                                    as_cmap=True)
             )

/Users/taugspurger/miniconda3/envs/travis-36-doc/lib/python3.6/site-packages/matplotlib/colors.py:504:
  xa[xa < 0] = -1

<pandas.io.formats.style.Styler at 0x1a15bdf8d0>

In [35]: def magnify():
         return [dict(selector="th",
                       props=[("font-size", "4pt")]),
                 dict(selector="td",
                       props=[('padding', "0em 0em")]),
                 dict(selector="th:hover",
                       props=[("font-size", "12pt")]),
                 dict(selector="tr:hover td:hover",
                       props=[('max-width', '200px'),
                              ('font-size', '12pt')])]

In [36]: np.random.seed(25)
         cmap = cmap=sns.diverging_palette(5, 250, as_cmap=True)
         bigdf = pd.DataFrame(np.random.randn(20, 25)).cumsum()

         bigdf.style.background_gradient(cmap, axis=1)\
             .set_properties(**{'max-width': '80px', 'font-size': '1pt'})\
             .set_caption("Hover to magnify")\
             .set_precision(2)\
             .set_table_styles(magnify())

Out[36]: <pandas.io.formats.style.Styler at 0x1a1d6d4978>
```

## 23.8 Export to Excel

*New in version 0.20.0*

*Experimental: This is a new feature and still under development. We'll be adding features and possibly making breaking changes in future releases. We'd love to hear your feedback.*

Some support is available for exporting styled DataFrames to Excel worksheets using the OpenPyXL or XlsxWriter engines. CSS2.2 properties handled include:

- background-color
- border-style, border-width, border-color and their {top, right, bottom, left variants}
- color
- font-family
- font-style
- font-weight
- text-align
- text-decoration



- vertical-align
- white-space: nowrap

Only CSS2 named colors and hex colors of the form #rgb or #rrggbb are currently supported.

```
In [37]: df.style.\
        applymap(color_negative_red).\
        apply(highlight_max).\
        to_excel('styled.xlsx', engine='openpyxl')
```

A screenshot of the output:

|    | A | B  | C         | D         | E         | F         |
|----|---|----|-----------|-----------|-----------|-----------|
| 1  |   | A  | B         | C         | D         | E         |
| 2  | 0 | 1  | 1.329212  |           | -0.31628  | -0.99081  |
| 3  | 1 | 2  | -1.070816 | -1.438713 | 0.564417  | 0.295722  |
| 4  | 2 | 3  | -1.626404 | 0.219565  | 0.678805  | 1.889273  |
| 5  | 3 | 4  | 0.961538  | 0.104011  | -0.481165 | 0.850229  |
| 6  | 4 | 5  | 1.453425  | 1.057737  | 0.165562  | 0.515018  |
| 7  | 5 | 6  | -1.336936 | 0.562861  | 1.392855  | -0.063328 |
| 8  | 6 | 7  | 0.121668  | 1.207603  | -0.00204  | 1.627796  |
| 9  | 7 | 8  | 0.354493  | 1.037528  | -0.385684 | 0.519818  |
| 10 | 8 | 9  | 1.686583  | -1.325963 | 1.428984  | -2.089354 |
| 11 | 9 | 10 | -0.12982  | 0.631523  | -0.586538 | 0.29072   |

Fig. 1: Excel spreadsheet with styled DataFrame

## 23.9 Extensibility

The core of pandas is, and will remain, its “high-performance, easy-to-use data structures”. With that in mind, we hope that `DataFrame.style` accomplishes two goals

- Provide an API that is pleasing to use interactively and is “good enough” for many tasks
- Provide the foundations for dedicated libraries to build on

If you build a great library on top of this, let us know and we’ll [link](#) to it.

### 23.9.1 Subclassing

If the default template doesn’t quite suit your needs, you can subclass `Styler` and extend or override the template. We’ll show an example of extending the default template to insert a custom header before each table.

```
In [38]: from jinja2 import Environment, ChoiceLoader, FileSystemLoader
        from IPython.display import HTML
        from pandas.io.formats.style import Styler
```

```
In [39]: %mkdir templates
```

This next cell writes the custom template. We extend the template `html.tpl`, which comes with pandas.

```
In [40]: %%file templates/myhtml.tpl
        {% extends "html.tpl" %}
        {% block table %}
        <h1>{{ table_title|default("My Table") }}</h1>
        {{ super() }}
        {% endblock table %}
```

Writing `templates/myhtml.tpl`

Now that we've created a template, we need to set up a subclass of `Styler` that knows about it.

```
In [41]: class MyStyler(Styler):
        env = Environment(
            loader=ChoiceLoader([
                FileSystemLoader("templates"), # contains ours
                Styler.loader, # the default
            ])
        )
        template = env.get_template("myhtml.tpl")
```

Notice that we include the original loader in our environment's loader. That's because we extend the original template, so the Jinja environment needs to be able to find it.

Now we can use that custom styler. It's `__init__` takes a `DataFrame`.

```
In [42]: MyStyler(df)
Out[42]: <__main__.MyStyler at 0x1a1d6d44e0>
```

Our custom template accepts a `table_title` keyword. We can provide the value in the `.render` method.

```
In [43]: HTML(MyStyler(df).render(table_title="Extending Example"))
Out[43]: <IPython.core.display.HTML object>
```

For convenience, we provide the `Styler.from_custom_template` method that does the same as the custom subclass.

```
In [44]: EasyStyler = Styler.from_custom_template("templates", "myhtml.tpl")
        EasyStyler(df)
Out[44]: <pandas.io.formats.style.Styler.from_custom_template.<locals>.MyStyler at 0x1a1fd934e0>
```

Here's the template structure:

```
In [45]: with open("template_structure.html") as f:
        structure = f.read()

        HTML(structure)
Out[45]: <IPython.core.display.HTML object>
```

See the template in the [GitHub repo](#) for more details.

## IO TOOLS (TEXT, CSV, HDF5, ...)

The pandas I/O API is a set of top level reader functions accessed like `pandas.read_csv()` that generally return a pandas object. The corresponding writer functions are object methods that are accessed like `DataFrame.to_csv()`. Below is a table containing available readers and writers.

| Format Type | Data Description     | Reader                      | Writer                    |
|-------------|----------------------|-----------------------------|---------------------------|
| text        | CSV                  | <code>read_csv</code>       | <code>to_csv</code>       |
| text        | JSON                 | <code>read_json</code>      | <code>to_json</code>      |
| text        | HTML                 | <code>read_html</code>      | <code>to_html</code>      |
| text        | Local clipboard      | <code>read_clipboard</code> | <code>to_clipboard</code> |
| binary      | MS Excel             | <code>read_excel</code>     | <code>to_excel</code>     |
| binary      | HDF5 Format          | <code>read_hdf</code>       | <code>to_hdf</code>       |
| binary      | Feather Format       | <code>read_feather</code>   | <code>to_feather</code>   |
| binary      | Parquet Format       | <code>read_parquet</code>   | <code>to_parquet</code>   |
| binary      | Msgpack              | <code>read_msgpack</code>   | <code>to_msgpack</code>   |
| binary      | Stata                | <code>read_stata</code>     | <code>to_stata</code>     |
| binary      | SAS                  | <code>read_sas</code>       |                           |
| binary      | Python Pickle Format | <code>read_pickle</code>    | <code>to_pickle</code>    |
| SQL         | SQL                  | <code>read_sql</code>       | <code>to_sql</code>       |
| SQL         | Google Big Query     | <code>read_gbq</code>       | <code>to_gbq</code>       |

*Here* is an informal performance comparison for some of these IO methods.

---

**Note:** For examples that use the `StringIO` class, make sure you import it according to your Python version, i.e. `from StringIO import StringIO` for Python 2 and `from io import StringIO` for Python 3.

---

### 24.1 CSV & Text files

The two workhorse functions for reading text files (a.k.a. flat files) are `read_csv()` and `read_table()`. They both use the same parsing code to intelligently convert tabular data into a `DataFrame` object. See the *cookbook* for some advanced strategies.

#### 24.1.1 Parsing options

The functions `read_csv()` and `read_table()` accept the following common arguments:

### 24.1.1.1 Basic

**filepath\_or\_buffer** [various] Either a path to a file (a `str`, `pathlib.Path`, or `py._path.local.LocalPath`), URL (including `http`, `ftp`, and `S3` locations), or any object with a `read()` method (such as an open file or `StringIO`).

**sep** [str, defaults to `,` for `read_csv()`, `\t` for `read_table()`] Delimiter to use. If `sep` is `None`, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\\r\\t'`.

**delimiter** [str, default `None`] Alternative argument name for `sep`.

**delim\_whitespace** [boolean, default `False`] Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the delimiter. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

### 24.1.1.2 Column and Index Locations and Names

**header** [int or list of ints, default `'infer'`] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names.

The header can be a list of ints that specify row locations for a multi-index on the columns e.g. `[0, 1, 3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

**names** [array-like, default `None`] List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

**index\_col** [int or sequence or `False`, default `None`] Column to use as the row labels of the `DataFrame`. If a sequence is given, a `MultiIndex` is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to *not* use the first column as the index (row names).

**usecols** [list-like or callable, default `None`] Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in `names` or inferred from the document header row(s). For example, a valid list-like `usecols` parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`.

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a `DataFrame` from data with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`:

```
In [1]: data = 'col1,col2,col3\na,b,1\na,b,2\nc,d,3'

In [2]: pd.read_csv(StringIO(data))
Out[2]:
   col1 col2 col3
```

(continues on next page)

(continued from previous page)

```

0    a    b    1
1    a    b    2
2    c    d    3

In [3]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['COL1', 'COL3
Out [3]:
   col1  col3
0     a     1
1     a     2
2     c     3

```

Using this parameter results in much faster parsing time and lower memory usage.

**squeeze** [boolean, default False] If the parsed data only contains one column then return a Series.

**prefix** [str, default None] Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, ...

**mangle\_dupe\_cols** [boolean, default True] Duplicate columns will be specified as 'X', 'X.1'... 'X.N', rather than 'X'... 'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

### 24.1.1.3 General Parsing Configuration

**dtype** [Type name or dict of column -> type, default None] Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} (unsupported with engine='python'). Use *str* or *object* together with suitable *na\_values* settings to preserve and not interpret dtype.

New in version 0.20.0: support for the Python parser.

**engine** [{'c', 'python'}] Parser engine to use. The C engine is faster while the Python engine is currently more feature-complete.

**converters** [dict, default None] Dict of functions for converting values in certain columns. Keys can either be integers or column labels.

**true\_values** [list, default None] Values to consider as True.

**false\_values** [list, default None] Values to consider as False.

**skipinitialspace** [boolean, default False] Skip spaces after delimiter.

**skiprows** [list-like or integer, default None] Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise:

```

In [4]: data = 'col1,col2,col3\na,b,1\na,b,2\nc,d,3'

In [5]: pd.read_csv(StringIO(data))
Out [5]:
   col1 col2 col3
0     a     b     1
1     a     b     2
2     c     d     3

In [6]: pd.read_csv(StringIO(data), skiprows=lambda x: x % 2 != 0)
Out [6]:

```

(continues on next page)

(continued from previous page)

|   | col1 | col2 | col3 |
|---|------|------|------|
| 0 | a    | b    | 2    |

**skipfooter** [int, default 0] Number of lines at bottom of file to skip (unsupported with engine='c').

**nrows** [int, default None] Number of rows of file to read. Useful for reading pieces of large files.

**low\_memory** [boolean, default True] Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set `False`, or specify the type with the `dtype` parameter. Note that the entire file is read into a single `DataFrame` regardless, use the `chunksize` or `iterator` parameter to return the data in chunks. (Only valid with C parser)

**memory\_map** [boolean, default False] If a filepath is provided for `filepath_or_buffer`, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

#### 24.1.1.4 NA and Missing Data Handling

**na\_values** [scalar, str, list-like, or dict, default None] Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. See [na\\_values const](#) below for a list of the values interpreted as NaN by default.

**keep\_default\_na** [boolean, default True] Whether or not to include the default NaN values when parsing the data. Depending on whether `na_values` is passed in, the behavior is as follows:

- If `keep_default_na` is `True`, and `na_values` are specified, `na_values` is appended to the default NaN values used for parsing.
- If `keep_default_na` is `True`, and `na_values` are not specified, only the default NaN values are used for parsing.
- If `keep_default_na` is `False`, and `na_values` are specified, only the NaN values specified `na_values` are used for parsing.
- If `keep_default_na` is `False`, and `na_values` are not specified, no strings will be parsed as NaN.

Note that if `na_filter` is passed in as `False`, the `keep_default_na` and `na_values` parameters will be ignored.

**na\_filter** [boolean, default True] Detect missing value markers (empty strings and the value of `na_values`). In data without any NAs, passing `na_filter=False` can improve the performance of reading a large file.

**verbose** [boolean, default False] Indicate number of NA values placed in non-numeric columns.

**skip\_blank\_lines** [boolean, default True] If `True`, skip over blank lines rather than interpreting as NaN values.

#### 24.1.1.5 Datetime Handling

**parse\_dates** [boolean or list of ints or names or list of lists or dict, default `False`.]

- If `True` -> try parsing the index.
- If `[1, 2, 3]` -> try parsing columns 1, 2, 3 each as a separate date column.
- If `[[1, 3]]` -> combine columns 1 and 3 and parse as a single date column.
- If `{'foo': [1, 3]}` -> parse columns 1, 3 as date and call result 'foo'. A fast-path exists for iso8601-formatted dates.

**infer\_datetime\_format** [boolean, default `False`] If `True` and `parse_dates` is enabled for a column, attempt to infer the datetime format to speed up the processing.

**keep\_date\_col** [boolean, default `False`] If `True` and `parse_dates` specifies combining multiple columns then keep the original columns.

**date\_parser** [function, default `None`] Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call `date_parser` in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by `parse_dates`) as arguments; 2) concatenate (row-wise) the string values from the columns defined by `parse_dates` into a single array and pass that; and 3) call `date_parser` once for each row using one or more strings (corresponding to the columns defined by `parse_dates`) as arguments.

**dayfirst** [boolean, default `False`] DD/MM format dates, international and European format.

#### 24.1.1.6 Iteration

**iterator** [boolean, default `False`] Return *TextFileReader* object for iteration or getting chunks with `get_chunk()`.

**chunksize** [int, default `None`] Return *TextFileReader* object for iteration. See *iterating and chunking* below.

#### 24.1.1.7 Quoting, Compression, and File Format

**compression** [{`'infer'`, `'gzip'`, `'bz2'`, `'zip'`, `'xz'`, `None`}, default `'infer'`] For on-the-fly decompression of on-disk data. If `'infer'`, then use `gzip`, `bz2`, `zip`, or `xz` if `filepath_or_buffer` is a string ending in `'gz'`, `'bz2'`, `'zip'`, or `'xz'`, respectively, and no decompression otherwise. If using `'zip'`, the ZIP file must contain only one data file to be read in. Set to `None` for no decompression.

New in version 0.18.1: support for `'zip'` and `'xz'` compression.

**thousands** [str, default `None`] Thousands separator.

**decimal** [str, default `'.'`] Character to recognize as decimal point. E.g. use `'.'` for European data.

**float\_precision** [string, default `None`] Specifies which converter the C engine should use for floating-point values. The options are `None` for the ordinary converter, `high` for the high-precision converter, and `round_trip` for the round-trip converter.

**lineterminator** [str (length 1), default `None`] Character to break file into lines. Only valid with C parser.

**quotechar** [str (length 1)] The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

**quoting** [int or `csv.QUOTE_*` instance, default 0] Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

**doublequote** [boolean, default `True`] When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements **inside** a field as a single `quotechar` element.

**escapechar** [str (length 1), default `None`] One-character string used to escape delimiter when quoting is `QUOTE_NONE`.

**comment** [str, default `None`] Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing `'#empty\na,b,c\n1,2,3'` with `header=0` will result in `'a,b,c'` being treated as the header.

**encoding** [str, default `None`] Encoding to use for UTF when reading/writing (e.g. `'utf-8'`). [List of Python standard encodings](#).

**dialect** [str or `csv.Dialect` instance, default `None`] If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a `ParserWarning` will be issued. See `csv.Dialect` documentation for more details.

**tupleize\_cols** [boolean, default `False`]

Deprecated since version 0.21.0.

This argument will be removed and will always convert to `MultiIndex`

Leave a list of tuples on columns as is (default is to convert to a `MultiIndex` on the columns).

#### 24.1.1.8 Error Handling

**error\_bad\_lines** [boolean, default `True`] Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no `DataFrame` will be returned. If `False`, then these “bad lines” will be dropped from the `DataFrame` that is returned. See *bad lines* below.

**warn\_bad\_lines** [boolean, default `True`] If `error_bad_lines` is `False`, and `warn_bad_lines` is `True`, a warning for each “bad line” will be output.

### 24.1.2 Specifying column data types

You can indicate the data type for the whole `DataFrame` or individual columns:

```
In [7]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'

In [8]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [9]: df = pd.read_csv(StringIO(data), dtype=object)

In [10]: df
Out[10]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

In [11]: df['a'][0]
Out[11]: '1'

In [12]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})

In [13]: df.dtypes
Out[13]:
a      int64
b      object
c      float64
dtype: object
```

Fortunately, pandas offers more than one way to ensure that your column(s) contain only one dtype. If you're unfamiliar with these concepts, you can see [here](#) to learn more about dtypes, and [here](#) to learn more about object conversion in pandas.



For instance, you can use the `converters` argument of `read_csv()`:

```
In [14]: data = "col_1\n1\n2\n'n'A'\n4.22"

In [15]: df = pd.read_csv(StringIO(data), converters={'col_1': str})

In [16]: df
Out[16]:
   col_1
0      1
1      2
2    'A'
3  4.22

In [17]: df['col_1'].apply(type).value_counts()
Out[17]:
<class 'str'>      4
Name: col_1, dtype: int64
```

Or you can use the `to_numeric()` function to coerce the dtypes after reading in the data,

```
In [18]: df2 = pd.read_csv(StringIO(data))

In [19]: df2['col_1'] = pd.to_numeric(df2['col_1'], errors='coerce')

In [20]: df2
Out[20]:
   col_1
0    1.00
1    2.00
2    NaN
3    4.22

In [21]: df2['col_1'].apply(type).value_counts()
Out[21]:
<class 'float'>      4
Name: col_1, dtype: int64
```

which will convert all valid parsing to floats, leaving the invalid parsing as NaN.

Ultimately, how you deal with reading in columns containing mixed dtypes depends on your specific needs. In the case above, if you wanted to NaN out the data anomalies, then `to_numeric()` is probably your best option. However, if you wanted for all the data to be coerced, no matter the type, then using the `converters` argument of `read_csv()` would certainly be worth trying.

New in version 0.20.0: support for the Python parser.

The `dtype` option is supported by the ‘python’ engine.

**Note:** In some cases, reading in abnormal data with columns containing mixed dtypes will result in an inconsistent dataset. If you rely on pandas to infer the dtypes of your columns, the parsing engine will go and infer the dtypes for different chunks of the data, rather than the whole dataset at once. Consequently, you can end up with column(s) with mixed dtypes. For example,

```
In [22]: df = pd.DataFrame({'col_1': list(range(500000)) + ['a', 'b'] +
↪ list(range(500000))})

In [23]: df.to_csv('foo.csv')
```

(continues on next page)

(continued from previous page)

```

In [24]: mixed_df = pd.read_csv('foo.csv')

In [25]: mixed_df['col_1'].apply(type).value_counts()
Out[25]:
<class 'int'>      737858
<class 'str'>      262144
Name: col_1, dtype: int64

In [26]: mixed_df['col_1'].dtype
Out[26]:
dtype('O')

```

will result with *mixed\_df* containing an `int` dtype for certain chunks of the column, and `str` for others due to the mixed dtypes from the data that was read in. It is important to note that the overall column will be marked with a dtype of `object`, which is used for columns with mixed dtypes.

### 24.1.3 Specifying Categorical dtype

New in version 0.19.0.

Categorical columns can be parsed directly by specifying `dtype='category'` or `dtype=CategoricalDtype(categories, ordered)`.

```

In [27]: data = 'col1,col2,col3\na,b,1\na,b,2\nc,d,3'

In [28]: pd.read_csv(StringIO(data))
Out[28]:
   col1 col2 col3
0     a    b    1
1     a    b    2
2     c    d    3

In [29]: pd.read_csv(StringIO(data)).dtypes
Out[29]:
col1    object
col2    object
col3    int64
dtype: object

In [30]: pd.read_csv(StringIO(data), dtype='category').dtypes
Out[30]:
col1    category
col2    category
col3    category
dtype: object

```

Individual columns can be parsed as a `Categorical` using a dict specification:

```

In [31]: pd.read_csv(StringIO(data), dtype={'col1': 'category'}).dtypes
Out[31]:
col1    category

```

(continues on next page)

(continued from previous page)

```
col2      object
col3      int64
dtype: object
```

New in version 0.21.0.

Specifying `dtype='category'` will result in an unordered `Categorical` whose categories are the unique values observed in the data. For more control on the categories and order, create a `CategoricalDtype` ahead of time, and pass that for that column's `dtype`.

```
In [32]: from pandas.api.types import CategoricalDtype

In [33]: dtype = CategoricalDtype(['d', 'c', 'b', 'a'], ordered=True)

In [34]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).dtypes
Out[34]:
col1      category
col2      object
col3      int64
dtype: object
```

When using `dtype=CategoricalDtype`, “unexpected” values outside of `dtype.categories` are treated as missing values.

```
In [35]: dtype = CategoricalDtype(['a', 'b', 'd']) # No 'c'

In [36]: pd.read_csv(StringIO(data), dtype={'col1': dtype}).col1
Out[36]:
0      a
1      a
2     NaN
Name: col1, dtype: category
Categories (3, object): [a, b, d]
```

This matches the behavior of `Categorical.set_categories()`.

**Note:** With `dtype='category'`, the resulting categories will always be parsed as strings (object dtype). If the categories are numeric they can be converted using the `to_numeric()` function, or as appropriate, another converter such as `to_datetime()`.

When `dtype` is a `CategoricalDtype` with homogenous categories ( all numeric, all datetimes, etc.), the conversion is done automatically.

```
In [37]: df = pd.read_csv(StringIO(data), dtype='category')

In [38]: df.dtypes
Out[38]:
col1      category
col2      category
col3      category
dtype: object

In [39]: df['col3']
Out[39]:
0      1
1      2
```

(continues on next page)

(continued from previous page)

```
2      3
Name: col3, dtype: category
Categories (3, object): [1, 2, 3]

In [40]: df['col3'].cat.categories = pd.to_numeric(df['col3'].cat.categories)

In [41]: df['col3']
Out[41]:
0      1
1      2
2      3
Name: col3, dtype: category
Categories (3, int64): [1, 2, 3]
```

### 24.1.4 Naming and Using Columns

#### 24.1.4.1 Handling column names

A file may or may not have a header row. pandas assumes the first row should be used as the column names:

```
In [42]: data = 'a,b,c\n1,2,3\n4,5,6\n7,8,9'
```

```
In [43]: print(data)
```

```
a,b,c
1,2,3
4,5,6
7,8,9
```

```
In [44]: pd.read_csv(StringIO(data))
```

```
Out[44]:
```

|   | a | b | c |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 4 | 5 | 6 |
| 2 | 7 | 8 | 9 |

By specifying the `names` argument in conjunction with `header` you can indicate other names to use and whether or not to throw away the header row (if any):

```
In [45]: print(data)
a,b,c
1,2,3
4,5,6
7,8,9

In [46]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[46]:
      foo  bar  baz
0      1    2    3
1      4    5    6
2      7    8    9

In [47]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
```

(continues on next page)

(continued from previous page)

```

foo bar baz
0    a    b    c
1    1    2    3
2    4    5    6
3    7    8    9

```

If the header is in a row other than the first, pass the row number to `header`. This will skip the preceding rows:

```

In [48]: data = 'skip this skip it\na,b,c\n1,2,3\n4,5,6\n7,8,9'

In [49]: pd.read_csv(StringIO(data), header=1)
Out[49]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

```

**Note:** Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first nonblank line of the file, if column names are passed explicitly then the behavior is identical to `header=None`.

### 24.1.5 Duplicate names parsing

If the file or header contains duplicate names, pandas will by default distinguish between them so as to prevent overwriting data:

```

In [50]: data = 'a,b,a\n0,1,2\n3,4,5'

In [51]: pd.read_csv(StringIO(data))
Out[51]:
   a  b  a.1
0  0  1     2
1  3  4     5

```

There is no more duplicate data because `mangle_dupe_cols=True` by default, which modifies a series of duplicate columns 'X', ..., 'X' to become 'X', 'X.1', ..., 'X.N'. If `mangle_dupe_cols=False`, duplicate data can arise:

```

In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
Out[3]:
   a  b  a
0  2  1  2
1  5  4  5

```

To prevent users from encountering this problem with duplicate data, a `ValueError` exception is raised if `mangle_dupe_cols != True`:

```

In [2]: data = 'a,b,a\n0,1,2\n3,4,5'
In [3]: pd.read_csv(StringIO(data), mangle_dupe_cols=False)
...
ValueError: Setting mangle_dupe_cols=False is not supported yet

```

### 24.1.5.1 Filtering columns (*usecols*)

The *usecols* argument allows you to select any subset of the columns in a file, either using the column names, position numbers or a callable:

New in version 0.20.0: support for callable *usecols* arguments

```
In [52]: data = 'a,b,c,d\n1,2,3,foo\n4,5,6,bar\n7,8,9,baz'

In [53]: pd.read_csv(StringIO(data))
Out[53]:
   a  b  c  d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz

In [54]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[54]:
   b  d
0  2  foo
1  5  bar
2  8  baz

In [55]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[55]:
   a  c  d
0  1  3  foo
1  4  6  bar
2  7  9  baz

In [56]: pd.read_csv(StringIO(data), usecols=lambda x: x.upper() in ['A', 'C'])
Out[56]:
   a  c
0  1  3
1  4  6
2  7  9
```

The *usecols* argument can also be used to specify which columns not to use in the final result:

```
In [57]: pd.read_csv(StringIO(data), usecols=lambda x: x not in ['a', 'c'])
Out[57]:
   b  d
0  2  foo
1  5  bar
2  8  baz
```

In this case, the callable is specifying that we exclude the “a” and “c” columns from the output.

## 24.1.6 Comments and Empty Lines

### 24.1.6.1 Ignoring line comments and empty lines

If the *comment* parameter is specified, then completely commented lines will be ignored. By default, completely blank lines will be ignored as well.

```
In [58]: data = '\na,b,c\n \n# commented line\n1,2,3\n\n4,5,6'

In [59]: print(data)

a,b,c

# commented line
1,2,3

4,5,6

In [60]: pd.read_csv(StringIO(data), comment='#')
Out[60]:
   a  b  c
0  1  2  3
1  4  5  6
```

If `skip_blank_lines=False`, then `read_csv` will not ignore blank lines:

```
In [61]: data = 'a,b,c\n1,2,3\n\n4,5,6'

In [62]: pd.read_csv(StringIO(data), skip_blank_lines=False)
Out[62]:
   a  b  c
0 NaN NaN NaN
1 1.0 2.0 3.0
2 NaN NaN NaN
3 NaN NaN NaN
4 4.0 5.0 6.0
```

**Warning:** The presence of ignored lines might create ambiguities involving line numbers; the parameter `header` uses row numbers (ignoring commented/empty lines), while `skiprows` uses line numbers (including commented/empty lines):

```
In [63]: data = '#comment\na,b,c\nA,B,C\n1,2,3'

In [64]: pd.read_csv(StringIO(data), comment='#', header=1)
Out[64]:
   A  B  C
0  1  2  3

In [65]: data = 'A,B,C\n#comment\na,b,c\n1,2,3'

In [66]: pd.read_csv(StringIO(data), comment='#', skiprows=2)
Out[66]:
   a  b  c
0  1  2  3
```

If both `header` and `skiprows` are specified, `header` will be relative to the end of `skiprows`. For example:

```

In [67]: data = '# empty\n# second empty line\n# third empty' \
In [67]: 'line\nX,Y,Z\n1,2,3\nA,B,C\n1,2.,4.\n5.,NaN,10.0'

In [68]: print(data)
# empty
# second empty line
# third emptyline
X,Y,Z
1,2,3
A,B,C
1,2.,4.
5.,NaN,10.0

In [69]: pd.read_csv(StringIO(data), comment='#', skiprows=4, header=1)
Out[69]:
  A    B    C
0  1.0  2.0  4.0
1  5.0  NaN 10.0

```

### 24.1.6.2 Comments

Sometimes comments or meta data may be included in a file:

```

In [70]: print(open('tmp.csv').read())
ID,level,category
Patient1,123000,x # really unpleasant
Patient2,23000,y # wouldn't take his medicine
Patient3,1234018,z # awesome

```

By default, the parser includes the comments in the output:

```

In [71]: df = pd.read_csv('tmp.csv')

In [72]: df
Out[72]:
   ID    level category
0  Patient1  123000      x # really unpleasant
1  Patient2   23000      y # wouldn't take his medicine
2  Patient3 1234018      z # awesome

```

We can suppress the comments using the `comment` keyword:

```

In [73]: df = pd.read_csv('tmp.csv', comment='#')

In [74]: df
Out[74]:
   ID    level category
0  Patient1  123000      x
1  Patient2   23000      y
2  Patient3 1234018      z

```



### 24.1.7 Dealing with Unicode Data

The `encoding` argument should be used for encoded unicode data, which will result in byte strings being decoded to unicode in the result:

```
In [75]: data = b'word,length\nTr\x03\xa4umen,7\nGr\x03\xbc\x03\x9fe,5'.decode('utf8
↳').encode('latin-1')

In [76]: df = pd.read_csv(BytesIO(data), encoding='latin-1')

In [77]: df
Out[77]:
   word  length
0  Träumen      7
1   Grüße      5

In [78]: df['word'][1]
Out[78]: 'Größe'
```

Some formats which encode all characters as multiple bytes, like UTF-16, won't parse correctly at all without specifying the encoding. [Full list of Python standard encodings](#).

### 24.1.8 Index columns and trailing delimiters

If a file has one more column of data than the number of column names, the first column will be used as the `DataFrame`'s row names:

```
In [79]: data = 'a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [80]: pd.read_csv(StringIO(data))
Out[80]:
   a      b      c
4  apple bat   5.7
8  orange cow  10.0
```

```
In [81]: data = 'index,a,b,c\n4,apple,bat,5.7\n8,orange,cow,10'

In [82]: pd.read_csv(StringIO(data), index_col=0)
Out[82]:
   index      a      b      c
4      apple bat   5.7
8      orange cow  10.0
```

Ordinarily, you can achieve this behavior using the `index_col` option.

There are some exception cases when a file has been prepared with delimiters at the end of each data line, confusing the parser. To explicitly disable the index column inference and discard the last column, pass `index_col=False`:

```
In [83]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow,'

In [84]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,
```

(continues on next page)

(continued from previous page)

```
In [85]: pd.read_csv(StringIO(data))
\\Out [85]:
      a    b    c
4  apple bat NaN
8  orange cow NaN
```

```
In [86]: pd.read_csv(StringIO(data), index_col=False)
\\Out [86]:
      a    b    c
0  4  apple bat
1  8  orange cow
```

If a subset of data is being parsed using the `usecols` option, the `index_col` specification is based on that subset, not the original data.

```
In [87]: data = 'a,b,c\n4,apple,bat,\n8,orange,cow, '
```

```
In [88]: print(data)
a,b,c
4,apple,bat,
8,orange,cow,
```

```
In [89]: pd.read_csv(StringIO(data), usecols=['b', 'c'])
\\Out [89]:
      b    c
4  bat NaN
8  cow NaN
```

```
In [90]: pd.read_csv(StringIO(data), usecols=['b', 'c'], index_col=0)
\\Out [90]:
      b    c
4  bat NaN
8  cow NaN
```

## 24.1.9 Date Handling

### 24.1.9.1 Specifying Date Columns

To better facilitate working with datetime data, `read_csv()` and `read_table()` use the keyword arguments `parse_dates` and `date_parser` to allow users to specify a variety of columns and date/time formats to turn the input text data into datetime objects.

The simplest case is to just pass in `parse_dates=True`:

```
# Use a column as an index, and parse it as dates.
In [91]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [92]: df
Out [92]:
      A  B  C
date
2009-01-01  a  1  2
2009-01-02  b  3  4
2009-01-03  c  4  5
```

(continues on next page)

(continued from previous page)

# These are Python datetime objects

**In [93]:** df.index

```

DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype='datetime64[ns]',
              name='date', freq=None)

```

It is often the case that we may want to store date and time data separately, or store various date fields separately. the `parse_dates` keyword can be used to specify a combination of columns to parse the dates and/or times from.

You can specify a list of column lists to `parse_dates`, the resulting date columns will be prepended to the output (so as to not affect the existing column order) and the new column names will be the concatenation of the component column names:

**In [94]:** print(open('tmp.csv').read())

```

KORD,19990127, 19:00:00, 18:56:00, 0.8100
KORD,19990127, 20:00:00, 19:56:00, 0.0100
KORD,19990127, 21:00:00, 20:56:00, -0.5900
KORD,19990127, 21:00:00, 21:18:00, -0.9900
KORD,19990127, 22:00:00, 21:56:00, -0.5900
KORD,19990127, 23:00:00, 22:56:00, -0.5900

```

**In [95]:** df = pd.read\_csv('tmp.csv', header=None, parse\_dates=[[1, 2], [1, 3]])**In [96]:** df**Out[96]:**

|   | 1_2                 | 1_3                 | 0    | 4     |
|---|---------------------|---------------------|------|-------|
| 0 | 1999-01-27 19:00:00 | 1999-01-27 18:56:00 | KORD | 0.81  |
| 1 | 1999-01-27 20:00:00 | 1999-01-27 19:56:00 | KORD | 0.01  |
| 2 | 1999-01-27 21:00:00 | 1999-01-27 20:56:00 | KORD | -0.59 |
| 3 | 1999-01-27 21:00:00 | 1999-01-27 21:18:00 | KORD | -0.99 |
| 4 | 1999-01-27 22:00:00 | 1999-01-27 21:56:00 | KORD | -0.59 |
| 5 | 1999-01-27 23:00:00 | 1999-01-27 22:56:00 | KORD | -0.59 |

By default the parser removes the component date columns, but you can choose to retain them via the `keep_date_col` keyword:

```

In [97]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]],
      ....:                  keep_date_col=True)
      ....:

```

**In [98]:** df**Out[98]:**

|   | 1_2                 | 1_3                 | 0    | 1        | 2        | 3        | 4     |
|---|---------------------|---------------------|------|----------|----------|----------|-------|
| 0 | 1999-01-27 19:00:00 | 1999-01-27 18:56:00 | KORD | 19990127 | 19:00:00 | 18:56:00 | 0.81  |
| 1 | 1999-01-27 20:00:00 | 1999-01-27 19:56:00 | KORD | 19990127 | 20:00:00 | 19:56:00 | 0.01  |
| 2 | 1999-01-27 21:00:00 | 1999-01-27 20:56:00 | KORD | 19990127 | 21:00:00 | 20:56:00 | -0.59 |
| 3 | 1999-01-27 21:00:00 | 1999-01-27 21:18:00 | KORD | 19990127 | 21:00:00 | 21:18:00 | -0.99 |
| 4 | 1999-01-27 22:00:00 | 1999-01-27 21:56:00 | KORD | 19990127 | 22:00:00 | 21:56:00 | -0.59 |
| 5 | 1999-01-27 23:00:00 | 1999-01-27 22:56:00 | KORD | 19990127 | 23:00:00 | 22:56:00 | -0.59 |

Note that if you wish to combine multiple columns into a single date column, a nested list must be used. In other words, `parse_dates=[1, 2]` indicates that the second and third columns should each be parsed as separate date columns while `parse_dates=[[1, 2]]` means the two columns should be parsed into a single column.

You can also use a dict to specify custom name columns:

```
In [99]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [100]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [101]: df
Out[101]:
```

|   | nominal             | actual              | 0    | 4     |
|---|---------------------|---------------------|------|-------|
| 0 | 1999-01-27 19:00:00 | 1999-01-27 18:56:00 | KORD | 0.81  |
| 1 | 1999-01-27 20:00:00 | 1999-01-27 19:56:00 | KORD | 0.01  |
| 2 | 1999-01-27 21:00:00 | 1999-01-27 20:56:00 | KORD | -0.59 |
| 3 | 1999-01-27 21:00:00 | 1999-01-27 21:18:00 | KORD | -0.99 |
| 4 | 1999-01-27 22:00:00 | 1999-01-27 21:56:00 | KORD | -0.59 |
| 5 | 1999-01-27 23:00:00 | 1999-01-27 22:56:00 | KORD | -0.59 |

It is important to remember that if multiple text columns are to be parsed into a single date column, then a new column is prepended to the data. The `index_col` specification is based off of this new set of columns rather than the original data columns:

```
In [102]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [103]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
.....:                    index_col=0) # index is the nominal column
.....:

In [104]: df
Out[104]:
```

|                     | actual              | 0    | 4     |
|---------------------|---------------------|------|-------|
| nominal             |                     |      |       |
| 1999-01-27 19:00:00 | 1999-01-27 18:56:00 | KORD | 0.81  |
| 1999-01-27 20:00:00 | 1999-01-27 19:56:00 | KORD | 0.01  |
| 1999-01-27 21:00:00 | 1999-01-27 20:56:00 | KORD | -0.59 |
| 1999-01-27 21:00:00 | 1999-01-27 21:18:00 | KORD | -0.99 |
| 1999-01-27 22:00:00 | 1999-01-27 21:56:00 | KORD | -0.59 |
| 1999-01-27 23:00:00 | 1999-01-27 22:56:00 | KORD | -0.59 |

---

**Note:** If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `to_datetime()` after `pd.read_csv`.

---

---

**Note:** `read_csv` has a `fast_path` for parsing datetime strings in iso8601 format, e.g. “2000-01-01T00:01:02+00:00” and similar variations. If you can arrange for your data to store datetimes in this format, load times will be significantly faster, ~20x has been observed.

---

---

**Note:** When passing a dict as the `parse_dates` argument, the order of the columns prepended is not guaranteed, because *dict* objects do not impose an ordering on their keys. On Python 2.7+ you may use `collections.OrderedDict` instead of a regular *dict* if this matters to you. Because of this, when using a dict for ‘`parse_dates`’ in conjunction with the `index_col` argument, it’s best to specify `index_col` as a column label rather than as an index on the resulting frame.

---

### 24.1.9.2 Date Parsing Functions

Finally, the parser allows you to specify a custom `date_parser` function to take full advantage of the flexibility of the date parsing API:

```
In [105]: import pandas.io.date_converters as conv

In [106]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
.....:                    date_parser=conv.parse_date_time)
.....:

In [107]: df
Out[107]:
```

|   | nominal             | actual              | 0    | 4     |
|---|---------------------|---------------------|------|-------|
| 0 | 1999-01-27 19:00:00 | 1999-01-27 18:56:00 | KORD | 0.81  |
| 1 | 1999-01-27 20:00:00 | 1999-01-27 19:56:00 | KORD | 0.01  |
| 2 | 1999-01-27 21:00:00 | 1999-01-27 20:56:00 | KORD | -0.59 |
| 3 | 1999-01-27 21:00:00 | 1999-01-27 21:18:00 | KORD | -0.99 |
| 4 | 1999-01-27 22:00:00 | 1999-01-27 21:56:00 | KORD | -0.59 |
| 5 | 1999-01-27 23:00:00 | 1999-01-27 22:56:00 | KORD | -0.59 |

Pandas will try to call the `date_parser` function in three different ways. If an exception is raised, the next one is tried:

1. `date_parser` is first called with one or more arrays as arguments, as defined using *parse\_dates* (e.g., `date_parser(['2013', '2013'], ['1', '2'])`).
2. If #1 fails, `date_parser` is called with all the columns concatenated row-wise into a single array (e.g., `date_parser(['2013 1', '2013 2'])`).
3. If #2 fails, `date_parser` is called once for every row with one or more string arguments from the columns indicated with *parse\_dates* (e.g., `date_parser('2013', '1')` for the first row, `date_parser('2013', '2')` for the second, etc.).

Note that performance-wise, you should try these methods of parsing dates in order:

1. Try to infer the format using `infer_datetime_format=True` (see section below).
2. If you know the format, use `pd.to_datetime(): date_parser=lambda x: pd.to_datetime(x, format=...)`.
3. If you have a really non-standard format, use a custom `date_parser` function. For optimal performance, this should be vectorized, i.e., it should accept arrays as arguments.

You can explore the date parsing functionality in `date_converters.py` and add your own. We would love to turn this module into a community supported set of date/time parsers. To get you started, `date_converters.py` contains functions to parse dual date and time columns, year/month/day columns, and year/month/day/hour/minute/second columns. It also contains a `generic_parser` function so you can curry it with a function that deals with a single date rather than the entire array.

### 24.1.9.3 Inferring Datetime Format

If you have `parse_dates` enabled for some or all of your columns, and your datetime strings are all formatted the same way, you may get a large speed up by setting `infer_datetime_format=True`. If set, pandas will attempt to guess the format of your datetime strings, and then use a faster means of parsing the strings. 5-10x parsing speeds have been observed. pandas will fallback to the usual parsing if either the format cannot be guessed or the format that was guessed cannot properly parse the entire column of strings. So in general, `infer_datetime_format` should not have any negative consequences if enabled.

Here are some examples of datetime strings that can be guessed (All representing December 30th, 2011 at 00:00:00):

- “20111230”
- “2011/12/30”
- “20111230 00:00:00”
- “12/30/2011 00:00:00”
- “30/Dec/2011 00:00:00”
- “30/December/2011 00:00:00”

Note that `infer_datetime_format` is sensitive to `dayfirst`. With `dayfirst=True`, it will guess “01/12/2011” to be December 1st. With `dayfirst=False` (default) it will guess “01/12/2011” to be January 12th.

```
# Try to infer the format for the index column
In [108]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True,
.....:                    infer_datetime_format=True)
.....:

In [109]: df
Out[109]:
```

|            | A | B | C |
|------------|---|---|---|
| date       |   |   |   |
| 2009-01-01 | a | 1 | 2 |
| 2009-01-02 | b | 3 | 4 |
| 2009-01-03 | c | 4 | 5 |

#### 24.1.9.4 International Date Formats

While US date formats tend to be MM/DD/YYYY, many international formats use DD/MM/YYYY instead. For convenience, a `dayfirst` keyword is provided:

```
In [110]: print(open('tmp.csv').read())
date,value,cat
1/6/2000,5,a
2/6/2000,10,b
3/6/2000,15,c

In [111]: pd.read_csv('tmp.csv', parse_dates=[0])
Out[111]:
   date    value cat
0 2000-01-06      5  a
1 2000-02-06     10  b
2 2000-03-06     15  c

In [112]: pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])
Out[112]:
   date    value cat
0 2000-06-01      5  a
1 2000-06-02     10  b
2 2000-06-03     15  c
```

### 24.1.10 Specifying method for floating-point conversion

The parameter `float_precision` can be specified in order to use a specific floating-point converter during parsing with the C engine. The options are the ordinary converter, the high-precision converter, and the round-trip converter (which is guaranteed to round-trip values after writing to a file). For example:

```
In [113]: val = '0.3066101993807095471566981359501369297504425048828125'

In [114]: data = 'a,b,c\n1,2,{0}'.format(val)

In [115]: abs(pd.read_csv(StringIO(data), engine='c', float_precision=None)['c'][0] -
↳ float(val))
Out[115]: 1.1102230246251565e-16

In [116]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='high')['c'][0] -
↳ float(val))
Out[116]: 5.5511151231257827e-17

In [117]: abs(pd.read_csv(StringIO(data), engine='c', float_precision='round_trip')['c']
↳ [0] - float(val))
Out[117]: 0.0
```

### 24.1.11 Thousand Separators

For large numbers that have been written with a thousands separator, you can set the `thousands` keyword to a string of length 1 so that integers will be parsed correctly:

By default, numbers with a thousands separator will be parsed as strings:

```
In [118]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z

In [119]: df = pd.read_csv('tmp.csv', sep='|')

In [120]: df
Out[120]:
   ID    level category
0  Patient1   123,000      x
1  Patient2    23,000      y
2  Patient3  1,234,018      z

In [121]: df.level.dtype
Out[121]: dtype('O')
```

The `thousands` keyword allows integers to be parsed correctly:

```
In [122]: print(open('tmp.csv').read())
ID|level|category
Patient1|123,000|x
Patient2|23,000|y
Patient3|1,234,018|z
```

(continues on next page)

(continued from previous page)

```
In [123]: df = pd.read_csv('tmp.csv', sep='|', thousands=',')
```

```
In [124]: df
```

Out[124]:

|   | ID       | level   | category |
|---|----------|---------|----------|
| 0 | Patient1 | 123000  | x        |
| 1 | Patient2 | 23000   | y        |
| 2 | Patient3 | 1234018 | z        |

```
In [125]: df.level.dtype
```

```
dtype('int64')
```

### 24.1.12 NA Values

To control which values are parsed as missing values (which are signified by `NaN`), specify a string in `na_values`. If you specify a list of strings, then all values in it are considered to be missing values. If you specify a number (a `float`, like `5.0` or an `integer` like `5`), the corresponding equivalent values will also imply a missing value (in this case effectively `[5.0, 5]` are recognized as `NaN`).

To completely override the default values that are recognized as missing, specify `keep_default_na=False`.

The default NaN recognized values are ['-1.#IND', '1.#QNAN', '1.#IND', '-1.#QNAN', '#N/A N/A', '#N/A', 'N/A', 'n/a', 'NA', '#NA', 'NULL', 'null', 'NaN', '-NaN', 'nan', '-nan', ''].

Let us consider some examples:

```
read_csv(path, na_values=[5])
```

In the example above 5 and 5.0 will be recognized as NaN, in addition to the defaults. A string will first be interpreted as a numerical 5, then as a NaN.

```
read_csv(path, keep_default_na=False, na_values=[""])
```

Above, only an empty field will be recognized as NaN.

```
read_csv(path, keep_default_na=False, na_values=["NA", "0"])
```

Above, both NA and 0 as strings are NaN.

```
read_csv(path, na_values=["Nope"])
```

The default values, in addition to the string "Nope" are recognized as NaN.

### 24.1.13 Infinity

inf like values will be parsed as `np.inf` (positive infinity), and `-inf` as `-np.inf` (negative infinity). These will ignore the case of the value, meaning `Inf`, will also be parsed as `np.inf`.

### 24.1.14 Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a Series:



```

In [126]: print(open('tmp.csv').read())
level
Patient1,123000
Patient2,23000
Patient3,1234018

In [127]: output = pd.read_csv('tmp.csv', squeeze=True)

In [128]: output
Out[128]:
Patient1      123000
Patient2      23000
Patient3     1234018
Name: level, dtype: int64

In [129]: type(output)
////////////////////////////////////Out
↳pandas.core.series.Series

```

### 24.1.15 Boolean values

The common values `True`, `False`, `TRUE`, and `FALSE` are all recognized as boolean. Occasionally you might want to recognize other values as being boolean. To do this, use the `true_values` and `false_values` options as follows:

```

In [130]: data= 'a,b,c\n1,Yes,2\n3,No,4'

In [131]: print(data)
a,b,c
1,Yes,2
3,No,4

In [132]: pd.read_csv(StringIO(data))
Out[132]:
   a  b  c
0  1  Yes 2
1  3  No  4

In [133]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[133]:
   a    b  c
0  1  True 2
1  3 False 4

```

### 24.1.16 Handling “bad” lines

Some files may have malformed lines with too few fields or too many. Lines with too few fields will have NA values filled in the trailing fields. Lines with too many fields will raise an error by default:

```

In [27]: data = 'a,b,c\n1,2,3\n4,5,6,7\n8,9,10'

In [28]: pd.read_csv(StringIO(data))
-----
ParserError                                Traceback (most recent call last)
ParserError: Error tokenizing data. C error: Expected 3 fields in line 3, saw 4

```

You can elect to skip bad lines:

```
In [29]: pd.read_csv(StringIO(data), error_bad_lines=False)
Skipping line 3: expected 3 fields, saw 4

Out[29]:
   a  b  c
0  1  2  3
1  8  9 10
```

You can also use the `usecols` parameter to eliminate extraneous column data that appear in some lines but not others:

```
In [30]: pd.read_csv(StringIO(data), usecols=[0, 1, 2])

Out[30]:
   a  b  c
0  1  2  3
1  4  5  6
2  8  9 10
```

### 24.1.17 Dialect

The `dialect` keyword gives greater flexibility in specifying the file format. By default it uses the Excel dialect but you can specify either the dialect name or a `csv.Dialect` instance.

Suppose you had data with unenclosed quotes:

```
In [134]: print(data)
label1,label2,label3
index1,"a,c,e
index2,b,d,f
```

By default, `read_csv` uses the Excel dialect and treats the double quote as the quote character, which causes it to fail when it finds a newline before it finds the closing double quote.

We can get around this using `dialect`:

```
In [135]: dia = csv.excel()

In [136]: dia.quoting = csv.QUOTE_NONE

In [137]: pd.read_csv(StringIO(data), dialect=dia)
Out[137]:
      label1 label2 label3
index1      "a      c      e
index2      b      d      f
```

All of the dialect options can be specified separately by keyword arguments:

```
In [138]: data = 'a,b,c~1,2,3~4,5,6'

In [139]: pd.read_csv(StringIO(data), lineterminator='~')
Out[139]:
   a  b  c
0  1  2  3
1  4  5  6
```

Another common dialect option is `skipinitialspace`, to skip any whitespace after a delimiter:

```
In [140]: data = 'a, b, c\n1, 2, 3\n4, 5, 6'

In [141]: print(data)
a, b, c
1, 2, 3
4, 5, 6

In [142]: pd.read_csv(StringIO(data), skipinitialspace=True)
Out[142]:
   a  b  c
0  1  2  3
1  4  5  6
```

The parsers make every attempt to “do the right thing” and not be fragile. Type inference is a pretty big deal. If a column can be coerced to integer dtype without altering the contents, the parser will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

### 24.1.18 Quoting and Escape Characters

Quotes (and other escape characters) in embedded fields can be handled in any number of ways. One way is to use backslashes; to properly parse this data, you should pass the `escapechar` option:

```
In [143]: data = 'a,b\n"hello, \\"Bob\\", nice to see you",5'

In [144]: print(data)
a,b
"hello, \\"Bob\\", nice to see you",5

In [145]: pd.read_csv(StringIO(data), escapechar='\\')
Out[145]:
```

|   | a                             | b |
|---|-------------------------------|---|
| 0 | hello, "Bob", nice to see you | 5 |

### 24.1.19 Files with Fixed Width Columns

While `read_csv()` reads delimited data, the `read_fwf()` function works with data files that have known and fixed column widths. The function parameters to `read_fwf` are largely the same as `read_csv` with two extra parameters, and a different usage of the `delimiter` parameter:

- **colspecs**: A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value 'infer' can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data. Default behavior, if not specified, is to infer.
- **widths**: A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.
- **delimiter**: Characters to consider as filler characters in the fixed-width file. Can be used to specify the filler character of the fields if it is not spaces (e.g., '~').

Consider a typical fixed-width data file:

```
In [146]: print(open('bar.csv').read())
```

|        |            |            |         |
|--------|------------|------------|---------|
| id8141 | 360.242940 | 149.910199 | 11950.7 |
| id1594 | 444.953632 | 166.985655 | 11788.4 |
| id1849 | 364.136849 | 183.628767 | 11806.2 |
| id1230 | 413.836124 | 184.375703 | 11916.8 |
| id1948 | 502.953953 | 173.237159 | 12468.3 |

In order to parse this file into a DataFrame, we simply need to supply the column specifications to the `read_fwf` function along with the file name:

```
# Column specifications are a list of half-intervals
In [147]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]

In [148]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)

In [149]: df
Out[149]:
```

|        | 1          | 2          | 3       |
|--------|------------|------------|---------|
| 0      |            |            |         |
| id8141 | 360.242940 | 149.910199 | 11950.7 |
| id1594 | 444.953632 | 166.985655 | 11788.4 |
| id1849 | 364.136849 | 183.628767 | 11806.2 |
| id1230 | 413.836124 | 184.375703 | 11916.8 |
| id1948 | 502.953953 | 173.237159 | 12468.3 |

Note how the parser automatically picks column names `X.<column number>` when `header=None` argument is specified. Alternatively, you can supply just the column widths for contiguous columns:

```
# Widths are a list of integers
In [150]: widths = [6, 14, 13, 10]

In [151]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [152]: df
Out[152]:
```

|   | 0      | 1          | 2          | 3       |
|---|--------|------------|------------|---------|
| 0 | id8141 | 360.242940 | 149.910199 | 11950.7 |
| 1 | id1594 | 444.953632 | 166.985655 | 11788.4 |
| 2 | id1849 | 364.136849 | 183.628767 | 11806.2 |
| 3 | id1230 | 413.836124 | 184.375703 | 11916.8 |
| 4 | id1948 | 502.953953 | 173.237159 | 12468.3 |

The parser will take care of extra white spaces around the columns so it's ok to have extra separation between the columns in the file.

By default, `read_fwf` will try to infer the file's `colspecs` by using the first 100 rows of the file. It can do it only in cases when the columns are aligned and correctly separated by the provided `delimiter` (default delimiter is whitespace).

```
In [153]: df = pd.read_fwf('bar.csv', header=None, index_col=0)

In [154]: df
Out[154]:
```

|        | 1          | 2          | 3       |
|--------|------------|------------|---------|
| 0      |            |            |         |
| id8141 | 360.242940 | 149.910199 | 11950.7 |
| id1594 | 444.953632 | 166.985655 | 11788.4 |
| id1849 | 364.136849 | 183.628767 | 11806.2 |
| id1230 | 413.836124 | 184.375703 | 11916.8 |
| id1948 | 502.953953 | 173.237159 | 12468.3 |

New in version 0.20.0.

`read_fwf` supports the `dtype` parameter for specifying the types of parsed columns to be different from the inferred type.

```
In [155]: pd.read_fwf('bar.csv', header=None, index_col=0).dtypes
Out[155]:
1    float64
2    float64
3    float64
dtype: object

In [156]: pd.read_fwf('bar.csv', header=None, dtype={2: 'object'}).dtypes
Out[156]:
0    object
1    float64
2    object
3    float64
dtype: object
```

## 24.1.20 Indexes

### 24.1.20.1 Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

```
In [157]: print(open('foo.csv').read())
A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5
```

In this special case, `read_csv` assumes that the first column is to be used as the index of the `DataFrame`:

```
In [158]: pd.read_csv('foo.csv')
Out[158]:
      A  B  C
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

Note that the dates weren’t automatically parsed. In that case you would need to do as before:

```
In [159]: df = pd.read_csv('foo.csv', parse_dates=True)

In [160]: df.index
Out[160]: DatetimeIndex(['2009-01-01', '2009-01-02', '2009-01-03'], dtype=
    <object> 'datetime64[ns]', freq=None)
```

### 24.1.20.2 Reading an index with a `MultiIndex`

Suppose you have data indexed by two columns:

```
In [161]: print(open('data/minindex_ex.csv').read())
year, indiv, zit, xit
1977, "A", 1.2, .6
1977, "B", 1.5, .5
1977, "C", 1.7, .8
1978, "A", .2, .06
```

(continues on next page)

(continued from previous page)

```

1978, "B", .7, .2
1978, "C", .8, .3
1978, "D", .9, .5
1978, "E", 1.4, .9
1979, "C", .2, .15
1979, "D", .14, .05
1979, "E", .5, .15
1979, "F", 1.2, .5
1979, "G", 3.4, 1.9
1979, "H", 5.4, 2.7
1979, "I", 6.4, 1.2

```

The `index_col` argument to `read_csv` and `read_table` can take a list of column numbers to turn multiple columns into a `MultiIndex` for the index of the returned object:

```
In [162]: df = pd.read_csv("data/minindex_ex.csv", index_col=[0,1])
```

```
In [163]: df
```

```
Out[163]:
```

|      |       | zit  | xit  |
|------|-------|------|------|
| year | indiv |      |      |
| 1977 | A     | 1.20 | 0.60 |
|      | B     | 1.50 | 0.50 |
|      | C     | 1.70 | 0.80 |
| 1978 | A     | 0.20 | 0.06 |
|      | B     | 0.70 | 0.20 |
|      | C     | 0.80 | 0.30 |
|      | D     | 0.90 | 0.50 |
|      | E     | 1.40 | 0.90 |
| 1979 | C     | 0.20 | 0.15 |
|      | D     | 0.14 | 0.05 |
|      | E     | 0.50 | 0.15 |
|      | F     | 1.20 | 0.50 |
|      | G     | 3.40 | 1.90 |
|      | H     | 5.40 | 2.70 |
|      | I     | 6.40 | 1.20 |

```
In [164]: df.loc[1978]
```

```

////////////////////////////////////
↪
      zit    xit
indiv
A      0.2  0.06
B      0.7  0.20
C      0.8  0.30
D      0.9  0.50
E      1.4  0.90

```

### 24.1.20.3 Reading columns with a `MultiIndex`

By specifying list of row locations for the `header` argument, you can read in a `MultiIndex` for the columns. Specifying non-consecutive rows will skip the intervening rows.

```
In [165]: from pandas.util.testing import makeCustomDataframe as mkdf
```

(continues on next page)

(continued from previous page)

```
In [166]: df = mkdf(5, 3, r_idx_nlevels=2, c_idx_nlevels=4)
```

```
In [167]: df.to_csv('mi.csv')
```

```
In [168]: print(open('mi.csv').read())
```

```
C0,,C_l0_g0,C_l0_g1,C_l0_g2
C1,,C_l1_g0,C_l1_g1,C_l1_g2
C2,,C_l2_g0,C_l2_g1,C_l2_g2
C3,,C_l3_g0,C_l3_g1,C_l3_g2
R0,R1,,,
R_l0_g0,R_l1_g0,R0C0,R0C1,R0C2
R_l0_g1,R_l1_g1,R1C0,R1C1,R1C2
R_l0_g2,R_l1_g2,R2C0,R2C1,R2C2
R_l0_g3,R_l1_g3,R3C0,R3C1,R3C2
R_l0_g4,R_l1_g4,R4C0,R4C1,R4C2
```

```
In [169]: pd.read_csv('mi.csv', header=[0, 1, 2, 3], index_col=[0, 1])
```

|         |         |         |         |         |
|---------|---------|---------|---------|---------|
|         |         |         |         |         |
| C0      |         | C_l0_g0 | C_l0_g1 | C_l0_g2 |
| C1      |         | C_l1_g0 | C_l1_g1 | C_l1_g2 |
| C2      |         | C_l2_g0 | C_l2_g1 | C_l2_g2 |
| C3      |         | C_l3_g0 | C_l3_g1 | C_l3_g2 |
| R0      | R1      |         |         |         |
| R_l0_g0 | R_l1_g0 | R0C0    | R0C1    | R0C2    |
| R_l0_g1 | R_l1_g1 | R1C0    | R1C1    | R1C2    |
| R_l0_g2 | R_l1_g2 | R2C0    | R2C1    | R2C2    |
| R_l0_g3 | R_l1_g3 | R3C0    | R3C1    | R3C2    |
| R_l0_g4 | R_l1_g4 | R4C0    | R4C1    | R4C2    |

`read_csv` is also able to interpret a more common format of multi-columns indices.

```
In [170]: print(open('mi2.csv').read())
```

```
,a,a,a,b,c,c
,q,r,s,t,u,v
one,1,2,3,4,5,6
two,7,8,9,10,11,12
```

```
In [171]: pd.read_csv('mi2.csv', header=[0, 1], index_col=0)
```

```

Out [171]:
      a      b      c
q  r  s  t  u  v
one 1 2 3 4 5 6
two 7 8 9 10 11 12

```

Note: If an `index_col` is not specified (e.g. you don't have an index, or wrote it with `df.to_csv(..., index=False)`), then any names on the columns index will be *lost*.

### 24.1.21 Automatically “sniffing” the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files, as `pandas` uses the `csv.Sniffer` class of the `csv` module. For this, you have to specify `sep=None`.

```
In [172]: print(open('tmp2.csv').read())
:0:1:2:3
0:0.4691122999071863:-0.2828633443286633:-1.5090585031735124:-1.1356323710171934
1:1.2121120250208506:-0.17321464905330858:0.11920871129693428:-1.0442359662799567
2:-0.8618489633477999:-2.1045692188948086:-0.4949292740687813:1.071803807037338
3:0.7215551622443669:-0.7067711336300845:-1.0395749851146963:0.27185988554282986
4:-0.42497232978883753:0.567020349793672:0.27623201927771873:-1.0874006912859915
5:-0.6736897080883706:0.1136484096888855:-1.4784265524372235:0.5249876671147047
6:0.4047052186802365:0.5770459859204836:-1.7150020161146375:-1.0392684835147725
7:-0.3706468582364464:-1.1578922506419993:-1.344311812731667:0.8448851414248841
8:1.0757697837155533:-0.10904997528022223:1.6435630703622064:-1.4693879595399115
9:0.35702056413309086:-0.6746001037299882:-1.776903716971867:-0.9689138124473498
```

```
In [173]: pd.read_csv('tmp2.csv', sep=None, engine='python')
```

```

////////////////////////////////////
↪
   Unnamed: 0      0      1      2      3
0      0  0.469112 -0.282863 -1.509059 -1.135632
1      1  1.212112 -0.173215  0.119209 -1.044236
2      2 -0.861849 -2.104569 -0.494929  1.071804
3      3  0.721555 -0.706771 -1.039575  0.271860
4      4 -0.424972  0.567020  0.276232 -1.087401
5      5 -0.673690  0.113648 -1.478427  0.524988
6      6  0.404705  0.577046 -1.715002 -1.039268
7      7 -0.370647 -1.157892 -1.344312  0.844885
8      8  1.075770 -0.109050  1.643563 -1.469388
9      9  0.357021 -0.674600 -1.776904 -0.968914
```

## 24.1.22 Reading multiple files to create a single DataFrame

It's best to use `concat()` to combine multiple files. See the [cookbook](#) for an example.

## 24.1.23 Iterating through files chunk by chunk

Suppose you wish to iterate through a (potentially very large) file lazily rather than reading the entire file into memory, such as the following:

```
In [174]: print(open('tmp.csv').read())
|0|1|2|3
0|0.4691122999071863|-0.2828633443286633|-1.5090585031735124|-1.1356323710171934
1|1.2121120250208506|-0.17321464905330858|0.11920871129693428|-1.0442359662799567
2|-0.8618489633477999|-2.1045692188948086|-0.4949292740687813|1.071803807037338
3|0.7215551622443669|-0.7067711336300845|-1.0395749851146963|0.27185988554282986
4|-0.42497232978883753|0.567020349793672|0.27623201927771873|-1.0874006912859915
5|-0.6736897080883706|0.1136484096888855|-1.4784265524372235|0.5249876671147047
6|0.4047052186802365|0.5770459859204836|-1.7150020161146375|-1.0392684835147725
7|-0.3706468582364464|-1.1578922506419993|-1.344311812731667|0.8448851414248841
8|1.0757697837155533|-0.10904997528022223|1.6435630703622064|-1.4693879595399115
9|0.35702056413309086|-0.6746001037299882|-1.776903716971867|-0.9689138124473498
```

```
In [175]: table = pd.read_table('tmp.csv', sep='|')
```

(continues on next page)





### 24.1.24 Specifying the parser engine

Under the hood pandas uses a fast and efficient parser implemented in C as well as a Python implementation which is currently more feature-complete. Where possible pandas uses the C parser (specified as `engine='c'`), but may fall back to Python if C-unsupported options are specified. Currently, C-unsupported options include:

- `sep` other than a single character (e.g. regex separators)
- `skipfooter`
- `sep=None` with `delim_whitespace=False`

Specifying any of the above options will produce a `ParserWarning` unless the python engine is selected explicitly using `engine='python'`.

### 24.1.25 Reading remote files

You can pass in a URL to a CSV file:

```
df = pd.read_csv('https://download.bls.gov/pub/time.series/cu/cu.item',
                 sep='\t')
```

S3 URLs are handled as well:

```
df = pd.read_csv('s3://pandas-test/tips.csv')
```

### 24.1.26 Writing out Data

#### 24.1.26.1 Writing to CSV format

The `Series` and `DataFrame` objects have an instance method `to_csv` which allows storing the contents of the object as a comma-separated-values file. The function takes a number of arguments. Only the first is required.

- `path_or_buf`: A string path to the file to write or a `StringIO`
- `sep`: Field delimiter for the output file (default `“,”`)
- `na_rep`: A string representation of a missing value (default `“”`)
- `float_format`: Format string for floating point numbers
- `cols`: Columns to write (default `None`)
- `header`: Whether to write out the column names (default `True`)
- `index`: whether to write row (index) names (default `True`)
- `index_label`: Column label(s) for index column(s) if desired. If `None` (default), and `header` and `index` are `True`, then the index names are used. (A sequence should be given if the `DataFrame` uses `MultiIndex`).
- `mode`: Python write mode, default `“w”`
- `encoding`: a string representing the encoding to use if the contents are non-ASCII, for Python versions prior to 3
- `line_terminator`: Character sequence denoting line end (default `“\n”`)
- `quoting`: Set quoting rules as in `csv` module (default `csv.QUOTE_MINIMAL`). Note that if you have set a `float_format` then floats are converted to strings and `csv.QUOTE_NONNUMERIC` will treat them as non-numeric

- `quotechar`: Character used to quote fields (default `'`)
- `doublequote`: Control quoting of `quotechar` in fields (default `True`)
- `escapechar`: Character used to escape `sep` and `quotechar` when appropriate (default `None`)
- `chunksize`: Number of rows to write at a time
- `tupleize_cols`: If `False` (default), write as a list of tuples, otherwise write in an expanded line format suitable for `read_csv`
- `date_format`: Format string for datetime objects

#### 24.1.26.2 Writing a formatted string

The `DataFrame` object has an instance method `to_string` which allows control over the string representation of the object. All arguments are optional:

- `buf` default `None`, for example a `StringIO` object
- `columns` default `None`, which columns to write
- `col_space` default `None`, minimum width of each column.
- `na_rep` default `NaN`, representation of NA value
- `formatters` default `None`, a dictionary (by column) of functions each of which takes a single argument and returns a formatted string
- `float_format` default `None`, a function which takes a single (float) argument and returns a formatted string; to be applied to floats in the `DataFrame`.
- `sparsify` default `True`, set to `False` for a `DataFrame` with a hierarchical index to print every multiindex key at each row.
- `index_names` default `True`, will print the names of the indices
- `index` default `True`, will print the index (ie, row labels)
- `header` default `True`, will print the column labels
- `justify` default `left`, will print column headers left- or right-justified

The `Series` object also has a `to_string` method, but with only the `buf`, `na_rep`, `float_format` arguments. There is also a `length` argument which, if set to `True`, will additionally output the length of the `Series`.

## 24.2 JSON

Read and write JSON format files and strings.

### 24.2.1 Writing JSON

A `Series` or `DataFrame` can be converted to a valid JSON string. Use `to_json` with optional parameters:

- `path_or_buf`: the pathname or buffer to write the output This can be `None` in which case a JSON string is returned
- `orient`:  
**Series:**

- default is index
- allowed values are {split, records, index}

**DataFrame:**

- default is columns
- allowed values are {split, records, index, columns, values, table}

The format of the JSON string

|         |  |
|---------|--|
| split   | dict like {index -> [index], columns -> [columns], data -> [values]} |
| records | list like [{column -> value}, ... , {column -> value}]               |
| index   | dict like {index -> {column -> value}}                               |
| columns | dict like {column -> {index -> value}}                               |
| values  | just the values array  |

- `date_format`: string, type of date conversion, 'epoch' for timestamp, 'iso' for ISO8601.
- `double_precision`: The number of decimal places to use when encoding floating point values, default 10.
- `force_ascii`: force encoded string to be ASCII, default True.
- `date_unit`: The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us' or 'ns' for seconds, milliseconds, microseconds and nanoseconds respectively. Default 'ms'.
- `default_handler`: The handler to call if an object cannot otherwise be converted to a suitable format for JSON. Takes a single argument, which is the object to convert, and returns a serializable object.
- `lines`: If records orient, then will write each record per line as json.

Note NaN's, NaT's and None will be converted to null and datetime objects will be converted based on the `date_format` and `date_unit` parameters.

```
In [182]: dfj = pd.DataFrame(randn(5, 2), columns=list('AB'))

In [183]: json = dfj.to_json()

In [184]: json
Out[184]: '{"A":{"0":-1.2945235903,"1":0.2766617129,"2":-0.0139597524,"3":-0.
↪0061535699,"4":0.8957173022},"B":{"0":0.4137381054,"1":-0.472034511,"2":-0.
↪3625429925,"3":-0.923060654,"4":0.8052440254}}'
```

### 24.2.1.1 Orient Options

There are a number of different options for the format of the resulting JSON file / string. Consider the following DataFrame and Series:

```
In [185]: dfjo = pd.DataFrame(dict(A=range(1, 4), B=range(4, 7), C=range(7, 10)),
.....:                          columns=list('ABC'), index=list('xyz'))
.....:

In [186]: dfjo
Out[186]:
   A  B  C
x  1  4  7
y  2  5  8
z  3  6  9
```

(continues on next page)

(continued from previous page)

```
In [187]: sjo = pd.Series(dict(x=15, y=16, z=17), name='D')

In [188]: sjo
Out[188]:
x      15
y      16
z      17
Name: D, dtype: int64
```

**Column oriented** (the default for DataFrame) serializes the data as nested JSON objects with column labels acting as the primary index:

```
In [189]: dfjo.to_json(orient="columns")
Out[189]: '{"A":{"x":1,"y":2,"z":3},"B":{"x":4,"y":5,"z":6},"C":{"x":7,"y":8,"z":9}}'

# Not available for Series
```

**Index oriented** (the default for Series) similar to column oriented but the index labels are now primary:

```
In [190]: dfjo.to_json(orient="index")
Out[190]: '{"x":{"A":1,"B":4,"C":7},"y":{"A":2,"B":5,"C":8},"z":{"A":3,"B":6,"C":9}}'

In [191]: sjo.to_json(orient="index")
Out[191]:
\\Out[191]:
↪ '{"x":15,"y":16,"z":17}'
```

**Record oriented** serializes the data to a JSON array of column -> value records, index labels are not included. This is useful for passing DataFrame data to plotting libraries, for example the JavaScript library d3.js:

```
In [192]: dfjo.to_json(orient="records")
Out[192]: '[{"A":1,"B":4,"C":7}, {"A":2,"B":5,"C":8}, {"A":3,"B":6,"C":9}]'

In [193]: sjo.to_json(orient="records")
Out[193]:
↪ '[15,16,17]'
```

**Value oriented** is a bare-bones option which serializes to nested JSON arrays of values only, column and index labels are not included:

```
In [194]: dfjo.to_json(orient="values")
Out[194]: '[[1,4,7],[2,5,8],[3,6,9]]'

# Not available for Series
```

**Split oriented** serializes to a JSON object containing separate entries for values, index and columns. Name is also included for Series:

```
In [195]: dfjo.to_json(orient="split")
Out[195]: '{"columns":["A","B","C"],"index":["x","y","z"],"data":[[1,4,7],[2,5,8],[3,6,9]]}'

In [196]: sjo.to_json(orient="split")
Out[196]:
↪ '{"name":"D","index":["x","y","z"],"data":[15,16,17]}'
```

**Table oriented** serializes to the JSON [Table Schema](#), allowing for the preservation of metadata including but not limited to dtypes and index names.

---

**Note:** Any orient option that encodes to a JSON object will not preserve the ordering of index and column labels during round-trip serialization. If you wish to preserve label ordering use the *split* option as it uses ordered containers.

---

### 24.2.1.2 Date Handling

Writing in ISO date format:

```
In [197]: dfd = pd.DataFrame(randn(5, 2), columns=list('AB'))

In [198]: dfd['date'] = pd.Timestamp('20130101')

In [199]: dfd = dfd.sort_index(1, ascending=False)

In [200]: json = dfd.to_json(date_format='iso')

In [201]: json
Out[201]: '{"date":{"0":"2013-01-01T00:00:00.000Z","1":"2013-01-01T00:00:00.000Z","2":
↪ "2013-01-01T00:00:00.000Z","3":"2013-01-01T00:00:00.000Z","4":"2013-01-01T00:00:00.
↪ 000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.8138502857,"4
↪":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.1702987971,"3":0.
↪ 4108345112,"4":0.1320031703}}'
```

Writing in ISO date format, with microseconds:

```
In [202]: json = dfd.to_json(date_format='iso', date_unit='us')

In [203]: json
Out[203]: '{"date":{"0":"2013-01-01T00:00:00.000000Z","1":"2013-01-01T00:00:00.000000Z
↪","2":"2013-01-01T00:00:00.000000Z","3":"2013-01-01T00:00:00.000000Z","4":"2013-01-
↪ 01T00:00:00.000000Z"},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3
↪":0.8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.
↪ 1702987971,"3":0.4108345112,"4":0.1320031703}}'
```

Epoch timestamps, in seconds:

```
In [204]: json = dfd.to_json(date_format='epoch', date_unit='s')

In [205]: json
Out[205]: '{"date":{"0":1356998400,"1":1356998400,"2":1356998400,"3":1356998400,"4
↪":1356998400},"B":{"0":2.5656459463,"1":1.3403088498,"2":-0.2261692849,"3":0.
↪ 8138502857,"4":-0.8273169356},"A":{"0":-1.2064117817,"1":1.4312559863,"2":-1.
↪ 1702987971,"3":0.4108345112,"4":0.1320031703}}'
```

Writing to a file, with a date index and a date column:

```
In [206]: dfj2 = dfj.copy()

In [207]: dfj2['date'] = pd.Timestamp('20130101')

In [208]: dfj2['ints'] = list(range(5))

In [209]: dfj2['bools'] = True
```

(continues on next page)

(continued from previous page)

```

In [210]: dfj2.index = pd.date_range('20130101', periods=5)

In [211]: dfj2.to_json('test.json')

In [212]: open('test.json').read()
Out[212]: '{"A":{"1356998400000":-1.2945235903,"1357084800000":0.2766617129,
↪ "1357171200000":-0.0139597524,"1357257600000":-0.0061535699,"1357344000000":0.
↪ 8957173022},"B":{"1356998400000":0.4137381054,"1357084800000":-0.472034511,
↪ "1357171200000":-0.3625429925,"1357257600000":-0.923060654,"1357344000000":0.
↪ 8052440254},"date":{"1356998400000":1356998400000,"1357084800000":1356998400000,
↪ "1357171200000":1356998400000,"1357257600000":1356998400000,"1357344000000
↪ ":1356998400000},"ints":{"1356998400000":0,"1357084800000":1,"1357171200000":2,
↪ "1357257600000":3,"1357344000000":4},"bools":{"1356998400000":true,"1357084800000
↪ ":true,"1357171200000":true,"1357257600000":true,"1357344000000":true}}'

```

### 24.2.1.3 Fallback Behavior

If the JSON serializer cannot handle the container contents directly it will fall back in the following manner:

- if the dtype is unsupported (e.g. `np.complex`) then the `default_handler`, if provided, will be called for each value, otherwise an exception is raised.
- if an object is unsupported it will attempt the following:
  - check if the object has defined a `toDict` method and call it. A `toDict` method should return a dict which will then be JSON serialized.
  - invoke the `default_handler` if one was provided.
  - convert the object to a dict by traversing its contents. However this will often fail with an `OverflowError` or give unexpected results.

In general the best approach for unsupported objects or dtypes is to provide a `default_handler`. For example:

```

DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json() # raises

RuntimeError: Unhandled numpy dtype 15

```

can be dealt with by specifying a simple `default_handler`:

```

In [213]: pd.DataFrame([1.0, 2.0, complex(1.0, 2.0)]).to_json(default_handler=str)
Out[213]: '{"0":{"0":"(1+0j)","1":"(2+0j)","2":"(1+2j)"}'

```

## 24.2.2 Reading JSON

Reading a JSON string to pandas object can take a number of parameters. The parser will try to parse a `DataFrame` if `typ` is not supplied or is `None`. To explicitly force `Series` parsing, pass `typ=series`

- `filepath_or_buffer`: a **VALID** JSON string or file handle / `StringIO`. The string could be a URL. Valid URL schemes include `http`, `ftp`, `S3`, and `file`. For file URLs, a host is expected. For instance, a local file could be `file://localhost/path/to/table.json`
- `typ`: type of object to recover (series or frame), default 'frame'
- `orient`:

**Series :**

- default is `index`
- allowed values are `{split, records, index}`

**DataFrame**

- default is `columns`
- allowed values are `{split, records, index, columns, values, table}`

The format of the JSON string

|                      |  |
|----------------------|--|
| <code>split</code>   | dict like <code>{index -&gt; [index], columns -&gt; [columns], data -&gt; [values]}</code> |
| <code>records</code> | list like <code>[{column -&gt; value}, ... , {column -&gt; value}]</code>                  |
| <code>index</code>   | dict like <code>{index -&gt; {column -&gt; value}}</code>                                  |
| <code>columns</code> | dict like <code>{column -&gt; {index -&gt; value}}</code>                                  |
| <code>values</code>  | just the values array  |
| <code>table</code>   | adhering to the JSON <a href="#">Table Schema</a>  |

- `dtype` : if `True`, infer dtypes, if a dict of column to dtype, then use those, if `False`, then don't infer dtypes at all, default is `True`, apply only to the data.
- `convert_axes` : boolean, try to convert the axes to the proper dtypes, default is `True`
- `convert_dates` : a list of columns to parse for dates; If `True`, then try to parse date-like columns, default is `True`.
- `keep_default_dates` : boolean, default `True`. If parsing dates, then parse the default date-like columns.
- `numpy` : direct decoding to NumPy arrays. default is `False`; Supports numeric data only, although labels may be non-numeric. Also note that the JSON ordering **MUST** be the same for each term if `numpy=True`.
- `precise_float` : boolean, default `False`. Set to enable usage of higher precision (`strtod`) function when decoding string to double values. Default (`False`) is to use fast but less precise builtin functionality.
- `date_unit` : string, the timestamp unit to detect if converting dates. Default `None`. By default the timestamp precision will be detected, if this is not desired then pass one of `'s'`, `'ms'`, `'us'` or `'ns'` to force timestamp precision to seconds, milliseconds, microseconds or nanoseconds respectively.
- `lines` : reads file as one json object per line.
- `encoding` : The encoding to use to decode py3 bytes.
- `chunksize` : when used in combination with `lines=True`, return a `JsonReader` which reads in `chunksize` lines per iteration.

The parser will raise one of `ValueError/TypeError/AssertionError` if the JSON is not parseable.

If a non-default `orient` was used when encoding to JSON be sure to pass the same option here so that decoding produces sensible results, see [Orient Options](#) for an overview.

### 24.2.2.1 Data Conversion

The default of `convert_axes=True`, `dtype=True`, and `convert_dates=True` will try to parse the axes, and all of the data into appropriate types, including dates. If you need to override specific dtypes, pass a dict to `dtype`. `convert_axes` should only be set to `False` if you need to preserve string-like numbers (e.g. `'1'`, `'2'`) in an axes.



**Note:** Large integer values may be converted to dates if `convert_dates=True` and the data and / or column labels appear ‘date-like’. The exact threshold depends on the `date_unit` specified. ‘date-like’ means that the column label meets one of the following criteria:

- it ends with ‘\_at’
- it ends with ‘\_time’
- it begins with ‘timestamp’
- it is ‘modified’
- it is ‘date’

**Warning:** When reading JSON data, automatic coercing into dtypes has some quirks:

- an index can be reconstructed in a different order from serialization, that is, the returned order is not guaranteed to be the same as before serialization
- a column that was `float` data will be converted to `integer` if it can be done safely, e.g. a column of 1.
- `bool` columns will be converted to `integer` on reconstruction

Thus there are times where you may want to specify specific dtypes via the `dtype` keyword argument.

Reading from a JSON string:

```
In [214]: pd.read_json(json)
```

```
Out [214]:
```

|   | date       | B         | A         |
|---|------------|-----------|-----------|
| 0 | 2013-01-01 | 2.565646  | -1.206412 |
| 1 | 2013-01-01 | 1.340309  | 1.431256  |
| 2 | 2013-01-01 | -0.226169 | -1.170299 |
| 3 | 2013-01-01 | 0.813850  | 0.410835  |
| 4 | 2013-01-01 | -0.827317 | 0.132003  |

Reading from a file:

```
In [215]: pd.read_json('test.json')
```

```
Out [215]:
```

|            | A         | B         | date       | ints | bools |
|------------|-----------|-----------|------------|------|-------|
| 2013-01-01 | -1.294524 | 0.413738  | 2013-01-01 | 0    | True  |
| 2013-01-02 | 0.276662  | -0.472035 | 2013-01-01 | 1    | True  |
| 2013-01-03 | -0.013960 | -0.362543 | 2013-01-01 | 2    | True  |
| 2013-01-04 | -0.006154 | -0.923061 | 2013-01-01 | 3    | True  |
| 2013-01-05 | 0.895717  | 0.805244  | 2013-01-01 | 4    | True  |

Don’t convert any data (but still convert axes and dates):

```
In [216]: pd.read_json('test.json', dtype=object).dtypes
```

```
Out [216]:
```

```
A      object
B      object
date    object
ints    object
bools   object
dtype: object
```

Specify dtypes for conversion:

```
In [217]: pd.read_json('test.json', dtype={'A': 'float32', 'bools': 'int8'}).dtypes
Out[217]:
A                float32
B                float64
date            datetime64[ns]
ints            int64
bools           int8
dtype: object
```

Preserve string indices:

[illegible]

```
In [219]: si
```

```
Out [219]:
```

|   | 0   | 1   | 2   | 3   |
|---|-----|-----|-----|-----|
| 0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2 | 0.0 | 0.0 | 0.0 | 0.0 |
| 3 | 0.0 | 0.0 | 0.0 | 0.0 |

```
In [220]: si.index
```

```
In [221]: si.columns
```

[illegible]

```
In [222]: json = si.to_json()
```

```
In [223]: sij = pd.read_json(json, convert_axes=False)
```

```
In [224]: sij
```

Out [224] :

|   | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 |

```
In [225]: sij.index
```

```
Out[225]:  
↪ Index(['0', '1', '2', '3'], dtype='object')
```

```
In [226]: sij.columns
```

[illegible]

Dates written in nanoseconds need to be read back in nanoseconds:

```
In [227]: json = dfj2.to_json(date_unit='ns')
```

(continues on next page)

(continued from previous page)

```
# Try to parse timestamps as milliseconds -> Won't Work
In [228]: dfju = pd.read_json(json, date_unit='ms')

In [229]: dfju
Out[229]:
```

|                     | A         | B         | date                | ints | bools |
|---------------------|-----------|-----------|---------------------|------|-------|
| 1356998400000000000 | -1.294524 | 0.413738  | 1356998400000000000 | 0    | True  |
| 1357084800000000000 | 0.276662  | -0.472035 | 1356998400000000000 | 1    | True  |
| 1357171200000000000 | -0.013960 | -0.362543 | 1356998400000000000 | 2    | True  |
| 1357257600000000000 | -0.006154 | -0.923061 | 1356998400000000000 | 3    | True  |
| 1357344000000000000 | 0.895717  | 0.805244  | 1356998400000000000 | 4    | True  |

```
# Let pandas detect the correct precision
In [230]: dfju = pd.read_json(json)

In [231]: dfju
Out[231]:
```

|            | A         | B         | date       | ints | bools |
|------------|-----------|-----------|------------|------|-------|
| 2013-01-01 | -1.294524 | 0.413738  | 2013-01-01 | 0    | True  |
| 2013-01-02 | 0.276662  | -0.472035 | 2013-01-01 | 1    | True  |
| 2013-01-03 | -0.013960 | -0.362543 | 2013-01-01 | 2    | True  |
| 2013-01-04 | -0.006154 | -0.923061 | 2013-01-01 | 3    | True  |
| 2013-01-05 | 0.895717  | 0.805244  | 2013-01-01 | 4    | True  |

```
# Or specify that all timestamps are in nanoseconds
In [232]: dfju = pd.read_json(json, date_unit='ns')

In [233]: dfju
Out[233]:
```

|            | A         | B         | date       | ints | bools |
|------------|-----------|-----------|------------|------|-------|
| 2013-01-01 | -1.294524 | 0.413738  | 2013-01-01 | 0    | True  |
| 2013-01-02 | 0.276662  | -0.472035 | 2013-01-01 | 1    | True  |
| 2013-01-03 | -0.013960 | -0.362543 | 2013-01-01 | 2    | True  |
| 2013-01-04 | -0.006154 | -0.923061 | 2013-01-01 | 3    | True  |
| 2013-01-05 | 0.895717  | 0.805244  | 2013-01-01 | 4    | True  |

#### 24.2.2.2 The Numpy Parameter

---

**Note:** This supports numeric data only. Index and columns labels may be non-numeric, e.g. strings, dates etc.

---

If `numpy=True` is passed to `read_json` an attempt will be made to sniff an appropriate dtype during deserialization and to subsequently decode directly to NumPy arrays, bypassing the need for intermediate Python objects.

This can provide speedups if you are deserialising a large amount of numeric data:

```
In [234]: randfloats = np.random.uniform(-100, 1000, 10000)

In [235]: randfloats.shape = (1000, 10)

In [236]: dffloats = pd.DataFrame(randfloats, columns=list('ABCDEFGHIJ'))

In [237]: jsonfloats = dffloats.to_json()
```

```
In [238]: timeit pd.read_json(jsonfloats)
8.6 ms +- 165 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [239]: timeit pd.read_json(jsonfloats, numpy=True)
6.39 ms +- 289 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

The speedup is less noticeable for smaller datasets:

```
In [240]: jsonfloats = dffloats.head(100).to_json()
```

```
In [241]: timeit pd.read_json(jsonfloats)
6.05 ms +- 793 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [242]: timeit pd.read_json(jsonfloats, numpy=True)
4.72 ms +- 159 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

**Warning:** Direct NumPy decoding makes a number of assumptions and may fail or produce unexpected output if these assumptions are not satisfied:

- data is numeric.
- data is uniform. The dtype is sniffed from the first value decoded. A `ValueError` may be raised, or incorrect output may be produced if this condition is not satisfied.
- labels are ordered. Labels are only read from the first container, it is assumed that each subsequent row / column has been encoded in the same order. This should be satisfied if the data was encoded using `to_json` but may not be the case if the JSON is from another source.

### 24.2.3 Normalization

pandas provides a utility function to take a dict or list of dicts and *normalize* this semi-structured data into a flat table.

```
In [243]: from pandas.io.json import json_normalize
```

```
In [244]: data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
.....:           {'name': {'given': 'Mose', 'family': 'Regner'}},
.....:           {'id': 2, 'name': 'Faye Raker'}]
.....:
```

```
In [245]: json_normalize(data)
```

```
Out[245]:
```

|   | id  | name       | name.family | name.first | name.given | name.last |
|---|-----|------------|-------------|------------|------------|-----------|
| 0 | 1.0 | NaN        | NaN         | Coleen     | NaN        | Volk      |
| 1 | NaN | NaN        | Regner      | NaN        | Mose       | NaN       |
| 2 | 2.0 | Faye Raker | NaN         | NaN        | NaN        | NaN       |

```
In [246]: data = [{'state': 'Florida',
.....:             'shortname': 'FL',
.....:             'info': {
.....:                 'governor': 'Rick Scott'
.....:             }},
.....:             {'counties': [{'name': 'Dade', 'population': 12345},
.....:                           {'name': 'Broward', 'population': 40000},
```

(continues on next page)

(continued from previous page)

```

.....:         {'name': 'Palm Beach', 'population': 60000}}],
.....:     {'state': 'Ohio',
.....:      'shortname': 'OH',
.....:      'info': {
.....:          'governor': 'John Kasich'
.....:      }},
.....:     'counties': [{'name': 'Summit', 'population': 1234},
.....:                  {'name': 'Cuyahoga', 'population': 1337}]]]
.....:
In [247]: json_normalize(data, 'counties', ['state', 'shortname', ['info', 'governor
↪']])
Out[247]:

```

|   | name       | population | state   | shortname | info.governor |
|---|------------|------------|---------|-----------|---------------|
| 0 | Dade       | 12345      | Florida | FL        | Rick Scott    |
| 1 | Broward    | 40000      | Florida | FL        | Rick Scott    |
| 2 | Palm Beach | 60000      | Florida | FL        | Rick Scott    |
| 3 | Summit     | 1234       | Ohio    | OH        | John Kasich   |
| 4 | Cuyahoga   | 1337       | Ohio    | OH        | John Kasich   |

## 24.2.4 Line delimited json

New in version 0.19.0.

pandas is able to read and write line-delimited json files that are common in data processing pipelines using Hadoop or Spark.

New in version 0.21.0.

For line-delimited json files, pandas can also return an iterator which reads in `chunksize` lines at a time. This can be useful for large files or to read from a stream.

```

In [248]: jsonl = '''
.....:     {"a": 1, "b": 2}
.....:     {"a": 3, "b": 4}
.....: '''
.....:

In [249]: df = pd.read_json(jsonl, lines=True)

In [250]: df
Out[250]:
   a  b
0  1  2
1  3  4

In [251]: df.to_json(orient='records', lines=True)
Out[251]: '{"a":1,"b":2}\n{"a":3,"b":4}'

# reader is an iterator that returns `chunksize` lines each iteration
In [252]: reader = pd.read_json(StringIO(jsonl), lines=True, chunksize=1)

In [253]: reader
Out[253]: <pandas.io.json.json.JsonReader at 0x1c3049f208>

In [254]: for chunk in reader:

```

(continues on next page)

(continued from previous page)

```
.....: print(chunk)
.....:
////////////////////////////////////\Empty DataFrame
Columns: []
Index: []
   a  b
0  1  2
   a  b
1  3  4
```

### 24.2.5 Table Schema

New in version 0.20.0.

**Table Schema** is a spec for describing tabular datasets as a JSON object. The JSON includes information on the field names, types, and other attributes. You can use the `orient table` to build a JSON string with two fields, `schema` and `data`.

```
In [255]: df = pd.DataFrame(
.....:     {'A': [1, 2, 3],
.....:      'B': ['a', 'b', 'c'],
.....:      'C': pd.date_range('2016-01-01', freq='d', periods=3),
.....:     }, index=pd.Index(range(3), name='idx'))
```

```
In [256]: df
```

```
Out [256]:
```

|     | A | B | C          |
|-----|---|---|------------|
| idx |   |   |            |
| 0   | 1 | a | 2016-01-01 |
| 1   | 2 | b | 2016-01-02 |
| 2   | 3 | c | 2016-01-03 |

```
In [257]: df.to_json(orient='table', date_format="iso")
\\////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////\\
↪ '{"schema": {"fields": [{"name": "idx", "type": "integer"}, {"name": "A", "type": "integer"}
↪ , {"name": "B", "type": "string"}, {"name": "C", "type": "datetime"}], "primaryKey": ["idx"],
↪ "pandas_version": "0.20.0"}, "data": [{"idx": 0, "A": 1, "B": "a", "C": "2016-01-
↪ 01T00:00:00.000Z"}, {"idx": 1, "A": 2, "B": "b", "C": "2016-01-02T00:00:00.000Z"}, {"idx": 2,
↪ "A": 3, "B": "c", "C": "2016-01-03T00:00:00.000Z"}]}'
```

The `schema` field contains the `fields` key, which itself contains a list of column name to type pairs, including the `Index` or `MultiIndex` (see below for a list of types). The `schema` field also contains a `primaryKey` field if the (Multi)index is unique.

The second field, `data`, contains the serialized data with the `records` orient. The index is included, and any datetimes are ISO 8601 formatted, as required by the Table Schema spec.

The full list of types supported are described in the Table Schema spec. This table shows the mapping from pandas types:

| Pandas type     | Table Schema type |
|-----------------|-------------------|
| int64           | integer           |
| float64         | number            |
| bool            | boolean           |
| datetime64[ns]  | datetime          |
| timedelta64[ns] | duration          |
| categorical     | any               |
| object          | str               |

A few notes on the generated table schema:

- The `schema` object contains a `pandas_version` field. This contains the version of pandas' dialect of the schema, and will be incremented with each revision.
- All dates are converted to UTC when serializing. Even timezone naïve values, which are treated as UTC with an offset of 0.

```
In [258]: from pandas.io.json import build_table_schema

In [259]: s = pd.Series(pd.date_range('2016', periods=4))

In [260]: build_table_schema(s)
Out[260]:
{'fields': [{'name': 'index', 'type': 'integer'},
             {'name': 'values', 'type': 'datetime'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- datetimes with a timezone (before serializing), include an additional field `tz` with the time zone name (e.g. 'US/Central').

```
In [261]: s_tz = pd.Series(pd.date_range('2016', periods=12,
.....:                                     tz='US/Central'))
.....:

In [262]: build_table_schema(s_tz)
Out[262]:
{'fields': [{'name': 'index', 'type': 'integer'},
             {'name': 'values', 'type': 'datetime', 'tz': 'US/Central'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Periods are converted to timestamps before serialization, and so have the same behavior of being converted to UTC. In addition, periods will contain an additional field `freq` with the period's frequency, e.g. 'A-DEC'.

```
In [263]: s_per = pd.Series(1, index=pd.period_range('2016', freq='A-DEC',
.....:                                               periods=4))
.....:

In [264]: build_table_schema(s_per)
Out[264]:
{'fields': [{'name': 'index', 'type': 'datetime', 'freq': 'A-DEC'},
             {'name': 'values', 'type': 'integer'}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- Categoricals use the `any` type and an `enum` constraint listing the set of possible values. Additionally, an ordered field is included:

```
In [265]: s_cat = pd.Series(pd.Categorical(['a', 'b', 'a']))

In [266]: build_table_schema(s_cat)
Out[266]:
{'fields': [{'name': 'index', 'type': 'integer'},
             {'name': 'values',
              'type': 'any',
              'constraints': {'enum': ['a', 'b']},
              'ordered': False}],
 'primaryKey': ['index'],
 'pandas_version': '0.20.0'}
```

- A `primaryKey` field, containing an array of labels, is included *if the index is unique*:

```
In [267]: s_dupe = pd.Series([1, 2], index=[1, 1])

In [268]: build_table_schema(s_dupe)
Out[268]:
{'fields': [{'name': 'index', 'type': 'integer'},
             {'name': 'values', 'type': 'integer'}],
 'pandas_version': '0.20.0'}
```

- The `primaryKey` behavior is the same with `MultiIndexes`, but in this case the `primaryKey` is an array:

```
In [269]: s_multi = pd.Series(1, index=pd.MultiIndex.from_product([('a', 'b'),
.....:                                                             (0, 1)]))
.....:

In [270]: build_table_schema(s_multi)
Out[270]:
{'fields': [{'name': 'level_0', 'type': 'string'},
             {'name': 'level_1', 'type': 'integer'},
             {'name': 'values', 'type': 'integer'}],
 'primaryKey': FrozenList(['level_0', 'level_1']),
 'pandas_version': '0.20.0'}
```

- The default naming roughly follows these rules:
  - For series, the `object.name` is used. If that's none, then the name is `values`
  - For DataFrames, the stringified version of the column name is used
  - For Index (not `MultiIndex`), `index.name` is used, with a fallback to `index` if that is `None`.
  - For `MultiIndex`, `mi.names` is used. If any level has no name, then `level_<i>` is used.

New in version 0.23.0.

`read_json` also accepts `orient='table'` as an argument. This allows for the preservation of metadata such as dtypes and index names in a round-trippable manner.

```
In [271]: df = pd.DataFrame({'foo': [1, 2, 3, 4],
.....:                      'bar': ['a', 'b', 'c', 'd'],
.....:                      'baz': pd.date_range('2018-01-01', freq='d',
→ periods=4),
.....:                      'qux': pd.Categorical(['a', 'b', 'c', 'c'])
.....:                      }, index=pd.Index(range(4), name='idx'))
```

(continues on next page)



(continued from previous page)

```

.....:

In [272]: df
Out[272]:
   foo bar      baz qux
idx
0     1  a 2018-01-01  a
1     2  b 2018-01-02  b
2     3  c 2018-01-03  c
3     4  d 2018-01-04  c

In [273]: df.dtypes
////////////////////////////////////
↪
foo          int64
bar          object
baz  datetime64[ns]
qux          category
dtype: object

In [274]: df.to_json('test.json', orient='table')

In [275]: new_df = pd.read_json('test.json', orient='table')

In [276]: new_df
Out[276]:
   foo bar      baz qux
idx
0     1  a 2018-01-01  a
1     2  b 2018-01-02  b
2     3  c 2018-01-03  c
3     4  d 2018-01-04  c

In [277]: new_df.dtypes
////////////////////////////////////
↪
foo          int64
bar          object
baz  datetime64[ns]
qux          category
dtype: object

```

Please note that the literal string ‘index’ as the name of an *Index* is not round-trippable, nor are any names beginning with ‘level\_’ within a *MultiIndex*. These are used by default in *DataFrame.to\_json()* to indicate missing values and the subsequent read cannot distinguish the intent.

```

In [278]: df.index.name = 'index'

In [279]: df.to_json('test.json', orient='table')

In [280]: new_df = pd.read_json('test.json', orient='table')

In [281]: print(new_df.index.name)
None

```

## 24.3 HTML

### 24.3.1 Reading HTML Content

**Warning:** We **highly encourage** you to read the *HTML Table Parsing gotchas* below regarding the issues surrounding the BeautifulSoup4/html5lib/lxml parsers.

The top-level `read_html()` function can accept an HTML string/file/URL and will parse HTML tables into list of pandas DataFrames. Let's look at a few examples.

**Note:** `read_html` returns a list of DataFrame objects, even if there is only a single table contained in the HTML content.

Read a URL with no options:

```
In [282]: url = 'http://www.fdic.gov/bank/individual/failed/banklist.html'

In [283]: dfs = pd.read_html(url)

In [284]: dfs
Out[284]:
```

|                     | Bank Name   | ... |  |
|---------------------|---|-----|--|
| ↪Updated Date       |   |     |  |
| 0                   | Washington Federal Bank for Savings               | ... |  |
| ↪February 21, 2018  |   |     |  |
| 1                   | The Farmers and Merchants State Bank of Argonia   | ... |  |
| ↪February 21, 2018  |   |     |  |
| 2                   | Fayette County Bank                               | ... |  |
| ↪July 26, 2017      |   |     |  |
| 3                   | Guaranty Bank, (d/b/a BestBank in Georgia & Mi... | ... |  |
| ↪March 22, 2018     |   |     |  |
| 4                   | First NBC Bank                                    | ... |  |
| ↪December 5, 2017   |   |     |  |
| 5                   | Proficio Bank                                     | ... |  |
| ↪March 7, 2018      |   |     |  |
| 6                   | Seaway Bank and Trust Company                     | ... |  |
| ↪May 18, 2017       |   |     |  |
| ..                  | ...   | ... |  |
| ↪                   |   |     |  |
| 548                 | Hamilton Bank, NA En Espanol                      | ... |  |
| ↪September 21, 2015 |   |     |  |
| 549                 | Sinclair National Bank                            | ... |  |
| ↪October 6, 2017    |   |     |  |
| 550                 | Superior Bank, FSB                                | ... |  |
| ↪August 19, 2014    |   |     |  |
| 551                 | Malta National Bank                               | ... |  |
| ↪November 18, 2002  |   |     |  |
| 552                 | First Alliance Bank & Trust Co.                   | ... |  |
| ↪February 18, 2003  |   |     |  |
| 553                 | National State Bank of Metropolis                 | ... |  |
| ↪March 17, 2005     |   |     |  |
| 554                 | Bank of Honolulu                                  | ... |  |
| ↪March 17, 2005     |   |     |  |

(continues on next page)

(continued from previous page)

```
[555 rows x 7 columns]]
```

**Note:** The data from the above URL changes every Monday so the resulting data above and the data below may be slightly different.

Read in the content of the file from the above URL and pass it to `read_html` as a string:

```
In [285]: with open(file_path, 'r') as f:
.....:     dfs = pd.read_html(f.read())
.....:

In [286]: dfs
Out[286]:
```

|                    |  | Bank Name    | City | ... |  |
|--------------------|--|--------------|------|-----|--|
| ↪Closing Date      | Updated Date                             |              |      |     |  |
| 0                  | Banks of Wisconsin d/b/a Bank of Kenosha | Kenosha      | ...  |     |  |
| ↪May 31, 2013      | May 31, 2013                             |              |      |     |  |
| 1                  | Central Arizona Bank                     | Scottsdale   | ...  |     |  |
| ↪May 14, 2013      | May 20, 2013                             |              |      |     |  |
| 2                  | Sunrise Bank                             | Valdosta     | ...  |     |  |
| ↪May 10, 2013      | May 21, 2013                             |              |      |     |  |
| 3                  | Pisgah Community Bank                    | Asheville    | ...  |     |  |
| ↪May 10, 2013      | May 14, 2013                             |              |      |     |  |
| 4                  | Douglas County Bank                      | Douglasville | ...  |     |  |
| ↪April 26, 2013    | May 16, 2013                             |              |      |     |  |
| 5                  | Parkway Bank                             | Lenoir       | ...  |     |  |
| ↪April 26, 2013    | May 17, 2013                             |              |      |     |  |
| 6                  | Chipola Community Bank                   | Marianna     | ...  |     |  |
| ↪April 19, 2013    | May 16, 2013                             |              |      |     |  |
| ..                 | ...                                      | ...          | ...  |     |  |
| ↪                  | ...                                      | ...          | ...  |     |  |
| 498                | Hamilton Bank, NAE n Espanol             | Miami        | ...  |     |  |
| ↪January 11, 2002  | June 5, 2012                             |              |      |     |  |
| 499                | Sinclair National Bank                   | Gravette     | ...  |     |  |
| ↪September 7, 2001 | February 10, 2004                        |              |      |     |  |
| 500                | Superior Bank, FSB                       | Hinsdale     | ...  |     |  |
| ↪July 27, 2001     | June 5, 2012                             |              |      |     |  |
| 501                | Malta National Bank                      | Malta        | ...  |     |  |
| ↪ May 3, 2001      | November 18, 2002                        |              |      |     |  |
| 502                | First Alliance Bank & Trust Co.          | Manchester   | ...  |     |  |
| ↪February 2, 2001  | February 18, 2003                        |              |      |     |  |
| 503                | National State Bank of Metropolis        | Metropolis   | ...  |     |  |
| ↪December 14, 2000 | March 17, 2005                           |              |      |     |  |
| 504                | Bank of Honolulu                         | Honolulu     | ...  |     |  |
| ↪October 13, 2000  | March 17, 2005                           |              |      |     |  |

```
[505 rows x 7 columns]]
```

You can even pass in an instance of `StringIO` if you so desire:

```
In [287]: with open(file_path, 'r') as f:
.....:     sio = StringIO(f.read())
.....:
```

(continues on next page)

(continued from previous page)

```
In [288]: dfs = pd.read_html(sio)
```

```
In [289]: dfs
```

```
Out[289]:
```

```
[
  Closing Date      Updated Date      Bank Name      City      ...
0      Banks of Wisconsin d/b/a Bank of Kenosha      Kenosha      ...
  May 31, 2013      May 31, 2013
1      Central Arizona Bank      Scottsdale      ...
  May 14, 2013      May 20, 2013
2      Sunrise Bank      Valdosta      ...
  May 10, 2013      May 21, 2013
3      Pisgah Community Bank      Asheville      ...
  May 10, 2013      May 14, 2013
4      Douglas County Bank      Douglasville      ...
  April 26, 2013      May 16, 2013
5      Parkway Bank      Lenoir      ...
  April 26, 2013      May 17, 2013
6      Chipola Community Bank      Marianna      ...
  April 19, 2013      May 16, 2013
  ..      ...      ...      ...      ...
  498      Hamilton Bank, NAEEn Espanol      Miami      ...
  January 11, 2002      June 5, 2012
  499      Sinclair National Bank      Gravette      ...
  September 7, 2001      February 10, 2004
  500      Superior Bank, FSB      Hinsdale      ...
  July 27, 2001      June 5, 2012
  501      Malta National Bank      Malta      ...
  May 3, 2001      November 18, 2002
  502      First Alliance Bank & Trust Co.      Manchester      ...
  February 2, 2001      February 18, 2003
  503      National State Bank of Metropolis      Metropolis      ...
  December 14, 2000      March 17, 2005
  504      Bank of Honolulu      Honolulu      ...
  October 13, 2000      March 17, 2005

[505 rows x 7 columns]]
```

**Note:** The following examples are not run by the IPython evaluator due to the fact that having so many network-accessing functions slows down the documentation build. If you spot an error or an example that doesn't run, please do not hesitate to report it over on [pandas GitHub issues page](#).

Read a URL and match a table that contains specific text:

```
match = 'Metcalf Bank'
df_list = pd.read_html(url, match=match)
```

Specify a header row (by default <th> or <td> elements located within a <thead> are used to form the column index, if multiple rows are contained within <thead> then a multiindex is created); if specified, the header row is taken from the data minus the parsed header elements (<th> elements).

```
dfs = pd.read_html(url, header=0)
```

Specify an index column:

```
dfs = pd.read_html(url, index_col=0)
```

Specify a number of rows to skip:

```
dfs = pd.read_html(url, skiprows=0)
```

Specify a number of rows to skip using a list (`xrange` (Python 2 only) works as well):

```
dfs = pd.read_html(url, skiprows=range(2))
```

Specify an HTML attribute:

```
dfs1 = pd.read_html(url, attrs={'id': 'table'})
dfs2 = pd.read_html(url, attrs={'class': 'sortable'})
print(np.array_equal(dfs1[0], dfs2[0])) # Should be True
```

Specify values that should be converted to NaN:

```
dfs = pd.read_html(url, na_values=['No Acquirer'])
```

New in version 0.19.

Specify whether to keep the default set of NaN values:

```
dfs = pd.read_html(url, keep_default_na=False)
```

New in version 0.19.

Specify converters for columns. This is useful for numerical text data that has leading zeros. By default columns that are numerical are cast to numeric types and the leading zeros are lost. To avoid this, we can convert these columns to strings.

```
url_mcc = 'https://en.wikipedia.org/wiki/Mobile_country_code'
dfs = pd.read_html(url_mcc, match='Telekom Albania', header=0, converters={'MNC':
str})
```

New in version 0.19.

Use some combination of the above:

```
dfs = pd.read_html(url, match='Metcalfe Bank', index_col=0)
```

Read in pandas `to_html` output (with some loss of floating point precision):

```
df = pd.DataFrame(randn(2, 2))
s = df.to_html(float_format='{0:.40g}'.format)
dfin = pd.read_html(s, index_col=0)
```

The `lxml` backend will raise an error on a failed parse if that is the only parser you provide. If you only have a single parser you can provide just a string, but it is considered good practice to pass a list with one string if, for example, the function expects a sequence of strings. You may use:

```
dfs = pd.read_html(url, 'Metcalfe Bank', index_col=0, flavor=['lxml'])
```

Or you could pass `flavor='lxml'` without a list:

```
dfs = pd.read_html(url, 'Metcalfe Bank', index_col=0, flavor='lxml')
```

However, if you have `bs4` and `html5lib` installed and pass `None` or `['lxml', 'bs4']` then the parse will most likely succeed. Note that *as soon as a parse succeeds, the function will return*.

```
dfs = pd.read_html(url, 'Metcalf Bank', index_col=0, flavor=['lxml', 'bs4'])
```

### 24.3.2 Writing to HTML files

`DataFrame` objects have an instance method `to_html` which renders the contents of the `DataFrame` as an HTML table. The function arguments are as in the method `to_string` described above.

---

**Note:** Not all of the possible options for `DataFrame.to_html` are shown here for brevity's sake. See `to_html()` for the full set of options.

---

```
In [290]: df = pd.DataFrame(randn(2, 2))

In [291]: df
Out[291]:
```

|   | 0         | 1        |
|---|-----------|----------|
| 0 | -0.184744 | 0.496971 |
| 1 | -0.856240 | 1.857977 |

```
In [292]: print(df.to_html()) # raw html
\\////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////<table_
↪border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>
```

HTML:

The `columns` argument will limit the columns shown:

```
In [293]: print(df.to_html(columns=[0]))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
```

(continues on next page)

(continued from previous page)

```

    </tr>
</thead>
<tbody>
  <tr>
    <th>0</th>
    <td>-0.184744</td>
  </tr>
  <tr>
    <th>1</th>
    <td>-0.856240</td>
  </tr>
</tbody>
</table>

```

HTML:

`float_format` takes a Python callable to control the precision of floating point values:

```

In [294]: print(df.to_html(float_format='{0:.10f}'.format))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.1847438576</td>
      <td>0.4969711327</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.8562396763</td>
      <td>1.8579766508</td>
    </tr>
  </tbody>
</table>

```

HTML:

`bold_rows` will make the row labels bold by default, but you can turn that off:

```

In [295]: print(df.to_html(bold_rows=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>0</td>

```

(continues on next page)

(continued from previous page)

```

        <td>-0.184744</td>
        <td>0.496971</td>
    </tr>
    <tr>
        <td>1</td>
        <td>-0.856240</td>
        <td>1.857977</td>
    </tr>
</tbody>
</table>

```

The `classes` argument provides the ability to give the resulting HTML table CSS classes. Note that these classes are *appended* to the existing 'dataframe' class.

```

In [296]: print(df.to_html(classes=['awesome_table_class', 'even_more_awesome_class
→']))
<table border="1" class="dataframe awesome_table_class even_more_awesome_class">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>0</th>
      <th>1</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>-0.184744</td>
      <td>0.496971</td>
    </tr>
    <tr>
      <th>1</th>
      <td>-0.856240</td>
      <td>1.857977</td>
    </tr>
  </tbody>
</table>

```

Finally, the `escape` argument allows you to control whether the “<”, “>” and “&” characters escaped in the resulting HTML (by default it is `True`). So to get the HTML without escaped characters pass `escape=False`

```

In [297]: df = pd.DataFrame({'a': list('&<>'), 'b': randn(3)})

```

Escaped:

```

In [298]: print(df.to_html())
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>

```

(continues on next page)



(continued from previous page)

```

        <th>0</th>
        <td>&lt;</td>
        <td>-0.474063</td>
    </tr>
    <tr>
        <th>1</th>
        <td>&lt;</td>
        <td>-0.230305</td>
    </tr>
    <tr>
        <th>2</th>
        <td>>></td>
        <td>-0.400654</td>
    </tr>
</tbody>
</table>

```

**Not escaped:**

```

In [299]: print(df.to_html(escape=False))
<table border="1" class="dataframe">
  <thead>
    <tr style="text-align: right;">
      <th></th>
      <th>a</th>
      <th>b</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th>0</th>
      <td>&</td>
      <td>-0.474063</td>
    </tr>
    <tr>
      <th>1</th>
      <td><</td>
      <td>-0.230305</td>
    </tr>
    <tr>
      <th>2</th>
      <td>></td>
      <td>-0.400654</td>
    </tr>
  </tbody>
</table>

```

---

**Note:** Some browsers may not show a difference in the rendering of the previous two HTML tables.

---

### 24.3.3 HTML Table Parsing Gotchas

There are some versioning issues surrounding the libraries that are used to parse HTML tables in the top-level pandas `io` function `read_html`.

### Issues with `lxml`

- Benefits
  - `lxml` is very fast.
  - `lxml` requires Cython to install correctly.
- Drawbacks
  - `lxml` does *not* make any guarantees about the results of its parse *unless* it is given **strictly valid markup**.
  - In light of the above, we have chosen to allow you, the user, to use the `lxml` backend, but **this backend will use `html5lib` if `lxml` fails to parse**
  - It is therefore *highly recommended* that you install both **BeautifulSoup4** and **html5lib**, so that you will still get a valid result (provided everything else is valid) even if `lxml` fails.

### Issues with **BeautifulSoup4** using `lxml` as a backend

- The above issues hold here as well since **BeautifulSoup4** is essentially just a wrapper around a parser backend.

### Issues with **BeautifulSoup4** using `html5lib` as a backend

- Benefits
  - **html5lib** is far more lenient than `lxml` and consequently deals with *real-life markup* in a much saner way rather than just, e.g., dropping an element without notifying you.
  - **html5lib** *generates valid HTML5 markup from invalid markup automatically*. This is extremely important for parsing HTML tables, since it guarantees a valid document. However, that does NOT mean that it is “correct”, since the process of fixing markup does not have a single definition.
  - **html5lib** is pure Python and requires no additional build steps beyond its own installation.
- Drawbacks
  - The biggest drawback to using **html5lib** is that it is slow as molasses. However consider the fact that many tables on the web are not big enough for the parsing algorithm runtime to matter. It is more likely that the bottleneck will be in the process of reading the raw text from the URL over the web, i.e., IO (input-output). For very large tables, this might not be true.

## 24.4 Excel files

The `read_excel()` method can read Excel 2003 (`.xls`) and Excel 2007+ (`.xlsx`) files using the `xlrd` Python module. The `to_excel()` instance method is used for saving a `DataFrame` to Excel. Generally the semantics are similar to working with `csv` data. See the *cookbook* for some advanced strategies.

### 24.4.1 Reading Excel Files

In the most basic use-case, `read_excel` takes a path to an Excel file, and the `sheet_name` indicating which sheet to parse.

```
# Returns a DataFrame
read_excel('path_to_file.xls', sheet_name='Sheet1')
```

### 24.4.1.1 ExcelFile class

To facilitate working with multiple sheets from the same file, the `ExcelFile` class can be used to wrap the file and can be passed into `read_excel`. There will be a performance benefit for reading multiple sheets as the file is read into memory only once.

```
xlsx = pd.ExcelFile('path_to_file.xls')
df = pd.read_excel(xlsx, 'Sheet1')
```

The `ExcelFile` class can also be used as a context manager.

```
with pd.ExcelFile('path_to_file.xls') as xls:
    df1 = pd.read_excel(xls, 'Sheet1')
    df2 = pd.read_excel(xls, 'Sheet2')
```

The `sheet_names` property will generate a list of the sheet names in the file.

The primary use-case for an `ExcelFile` is parsing multiple sheets with different parameters:

```
data = {}
# For when Sheet1's format differs from Sheet2
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = pd.read_excel(xls, 'Sheet1', index_col=None, na_values=['NA'])
    data['Sheet2'] = pd.read_excel(xls, 'Sheet2', index_col=1)
```

Note that if the same parsing parameters are used for all sheets, a list of sheet names can simply be passed to `read_excel` with no loss in performance.

```
# using the ExcelFile class
data = {}
with pd.ExcelFile('path_to_file.xls') as xls:
    data['Sheet1'] = read_excel(xls, 'Sheet1', index_col=None, na_values=['NA'])
    data['Sheet2'] = read_excel(xls, 'Sheet2', index_col=None, na_values=['NA'])

# equivalent using the read_excel function
data = read_excel('path_to_file.xls', ['Sheet1', 'Sheet2'], index_col=None, na_
↪ values=['NA'])
```

### 24.4.1.2 Specifying Sheets

---

**Note:** The second argument is `sheet_name`, not to be confused with `ExcelFile.sheet_names`.

---



---

**Note:** An `ExcelFile`'s attribute `sheet_names` provides access to a list of sheets.

---

- The arguments `sheet_name` allows specifying the sheet or sheets to read.
- The default value for `sheet_name` is 0, indicating to read the first sheet
- Pass a string to refer to the name of a particular sheet in the workbook.
- Pass an integer to refer to the index of a sheet. Indices follow Python convention, beginning at 0.
- Pass a list of either strings or integers, to return a dictionary of specified sheets.
- Pass a `None` to return a dictionary of all available sheets.

```
# Returns a DataFrame
read_excel('path_to_file.xls', 'Sheet1', index_col=None, na_values=['NA'])
```

Using the sheet index:

```
# Returns a DataFrame
read_excel('path_to_file.xls', 0, index_col=None, na_values=['NA'])
```

Using all default values:

```
# Returns a DataFrame
read_excel('path_to_file.xls')
```

Using None to get all sheets:

```
# Returns a dictionary of DataFrames
read_excel('path_to_file.xls', sheet_name=None)
```

Using a list to get multiple sheets:

```
# Returns the 1st and 4th sheet, as a dictionary of DataFrames.
read_excel('path_to_file.xls', sheet_name=['Sheet1', 3])
```

`read_excel` can read more than one sheet, by setting `sheet_name` to either a list of sheet names, a list of sheet positions, or `None` to read all sheets. Sheets can be specified by sheet index or sheet name, using an integer or string, respectively.

#### 24.4.1.3 Reading a MultiIndex

`read_excel` can read a `MultiIndex` index, by passing a list of columns to `index_col` and a `MultiIndex` column by passing a list of rows to `header`. If either the index or columns have serialized level names those will be read in as well by specifying the rows/columns that make up the levels.

For example, to read in a `MultiIndex` index without names:

```
In [300]: df = pd.DataFrame({'a':[1, 2, 3, 4], 'b':[5, 6, 7, 8]},
.....:                      index=pd.MultiIndex.from_product([['a', 'b'], ['c', 'd']]))
.....:

In [301]: df.to_excel('path_to_file.xlsx')

In [302]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])

In [303]: df
Out[303]:
      a  b
a c   1  5
  d   2  6
b c   3  7
  d   4  8
```

If the index has level names, they will be parsed as well, using the same parameters.

```
In [304]: df.index = df.index.set_names(['lvl1', 'lvl2'])

In [305]: df.to_excel('path_to_file.xlsx')
```

(continues on next page)

(continued from previous page)

```
In [306]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1])
```

```
In [307]: df
```

```
Out[307]:
```

```

      a  b
lvl1 lvl2
a     c   1  5
      d   2  6
b     c   3  7
      d   4  8

```

If the source file has both MultiIndex index and columns, lists specifying each should be passed to `index_col` and `header`:

```
In [308]: df.columns = pd.MultiIndex.from_product([['a'], ['b', 'd']], names=['c1',
→ 'c2'])
```

```
In [309]: df.to_excel('path_to_file.xlsx')
```

```
In [310]: df = pd.read_excel('path_to_file.xlsx', index_col=[0, 1], header=[0, 1])
```

```
In [311]: df
```

```
Out[311]:
```

```

c1      a
c2      b  d
lvl1 lvl2
a     c   1  5
      d   2  6
b     c   3  7
      d   4  8

```

#### 24.4.1.4 Parsing Specific Columns

It is often the case that users will insert columns to do temporary computations in Excel and you may not want to read in those columns. `read_excel` takes a `usecols` keyword to allow you to specify a subset of columns to parse.

If `usecols` is an integer, then it is assumed to indicate the last column to be parsed.

```
read_excel('path_to_file.xls', 'Sheet1', usecols=2)
```

If `usecols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```
read_excel('path_to_file.xls', 'Sheet1', usecols=[0, 2, 3])
```

Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`.

#### 24.4.1.5 Parsing Dates

Datetime-like values are normally automatically converted to the appropriate dtype when reading the excel file. But if you have a column of strings that *look* like dates (but are not actually formatted as dates in excel), you can use the `parse_dates` keyword to parse those strings to datetimes:

```
read_excel('path_to_file.xls', 'Sheet1', parse_dates=['date_strings'])
```

#### 24.4.1.6 Cell Converters

It is possible to transform the contents of Excel cells via the `converters` option. For instance, to convert a column to boolean:

```
read_excel('path_to_file.xls', 'Sheet1', converters={'MyBools': bool})
```

This options handles missing values and treats exceptions in the converters as missing data. Transformations are applied cell by cell rather than to the column as a whole, so the array dtype is not guaranteed. For instance, a column of integers with missing values cannot be transformed to an array with integer dtype, because NaN is strictly a float. You can manually mask missing data to recover integer dtype:

```
cfun = lambda x: int(x) if x else -1
read_excel('path_to_file.xls', 'Sheet1', converters={'MyInts': cfun})
```

#### 24.4.1.7 dtype Specifications

New in version 0.20.

As an alternative to converters, the type for an entire column can be specified using the `dtype` keyword, which takes a dictionary mapping column names to types. To interpret data with no type inference, use the type `str` or `object`.

```
read_excel('path_to_file.xls', dtype={'MyInts': 'int64', 'MyText': str})
```

### 24.4.2 Writing Excel Files

#### 24.4.2.1 Writing Excel Files to Disk

To write a `DataFrame` object to a sheet of an Excel file, you can use the `to_excel` instance method. The arguments are largely the same as `to_csv` described above, the first argument being the name of the excel file, and the optional second argument the name of the sheet to which the `DataFrame` should be written. For example:

```
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

Files with a `.xls` extension will be written using `xlwt` and those with a `.xlsx` extension will be written using `xlsxwriter` (if available) or `openpyxl`.

The `DataFrame` will be written in a way that tries to mimic the REPL output. The `index_label` will be placed in the second row instead of the first. You can place it in the first row by setting the `merge_cells` option in `to_excel()` to `False`:

```
df.to_excel('path_to_file.xlsx', index_label='label', merge_cells=False)
```

In order to write separate `DataFrames` to separate sheets in a single Excel file, one can pass an `ExcelWriter`.

```
with ExcelWriter('path_to_file.xlsx') as writer:
    df1.to_excel(writer, sheet_name='Sheet1')
    df2.to_excel(writer, sheet_name='Sheet2')
```

---

**Note:** Wringing a little more performance out of `read_excel` Internally, Excel stores all numeric data as floats. Because this can produce unexpected behavior when reading in data, pandas defaults to trying to convert integers to floats if it doesn't lose information (`1.0 --> 1`). You can pass `convert_float=False` to disable this behavior, which may give a slight performance improvement.

---

### 24.4.2.2 Writing Excel Files to Memory

Pandas supports writing Excel files to buffer-like objects such as `StringIO` or `BytesIO` using `ExcelWriter`.

```
# Safe import for either Python 2.x or 3.x
try:
    from io import BytesIO
except ImportError:
    from cStringIO import StringIO as BytesIO

bio = BytesIO()

# By setting the 'engine' in the ExcelWriter constructor.
writer = ExcelWriter(bio, engine='xlsxwriter')
df.to_excel(writer, sheet_name='Sheet1')

# Save the workbook
writer.save()

# Seek to the beginning and read to copy the workbook to a variable in memory
bio.seek(0)
workbook = bio.read()
```

---

**Note:** `engine` is optional but recommended. Setting the engine determines the version of workbook produced. Setting `engine='xlrd'` will produce an Excel 2003-format workbook (xls). Using either `'openpyxl'` or `'xlsxwriter'` will produce an Excel 2007-format workbook (xlsx). If omitted, an Excel 2007-formatted workbook is produced.

---

### 24.4.3 Excel writer engines

Pandas chooses an Excel writer via two methods:

1. the `engine` keyword argument
2. the filename extension (via the default specified in config options)

By default, pandas uses the `XlsxWriter` for `.xlsx`, `openpyxl` for `.xlsm`, and `xlwt` for `.xls` files. If you have multiple engines installed, you can set the default engine through *setting the config options* `io.excel.xlsx.writer` and `io.excel.xls.writer`. pandas will fall back on `openpyxl` for `.xlsx` files if `Xlsxwriter` is not available.

To specify which writer you want to use, you can pass an engine keyword argument to `to_excel` and to `ExcelWriter`. The built-in engines are:

- `openpyxl`: version 2.4 or higher is required
- `xlsxwriter`
- `xlwt`

```
# By setting the 'engine' in the DataFrame and Panel 'to_excel()' methods.
df.to_excel('path_to_file.xlsx', sheet_name='Sheet1', engine='xlsxwriter')

# By setting the 'engine' in the ExcelWriter constructor.
writer = ExcelWriter('path_to_file.xlsx', engine='xlsxwriter')

# Or via pandas configuration.
from pandas import options
options.io.excel.xlsx.writer = 'xlsxwriter'

df.to_excel('path_to_file.xlsx', sheet_name='Sheet1')
```

### 24.4.4 Style and Formatting

The look and feel of Excel worksheets created from pandas can be modified using the following parameters on the DataFrame's `to_excel` method.

- `float_format` : Format string for floating point numbers (default `None`).
- `freeze_panes` : A tuple of two integers representing the bottommost row and rightmost column to freeze. Each of these parameters is one-based, so (1, 1) will freeze the first row and first column (default `None`).

## 24.5 Clipboard

A handy way to grab data is to use the `read_clipboard()` method, which takes the contents of the clipboard buffer and passes them to the `read_table` method. For instance, you can copy the following text to the clipboard (CTRL-C on many operating systems):

```
A B C
x 1 4 p
y 2 5 q
z 3 6 r
```

And then import the data directly to a DataFrame by calling:

```
clipdf = pd.read_clipboard()
```

```
In [312]: clipdf
Out[312]:
   A  B  C
x  1  4  p
y  2  5  q
z  3  6  r
```

The `to_clipboard` method can be used to write the contents of a DataFrame to the clipboard. Following which you can paste the clipboard contents into other applications (CTRL-V on many operating systems). Here we illustrate writing a DataFrame into clipboard and reading it back.

```
In [313]: df = pd.DataFrame(randn(5, 3))

In [314]: df
Out[314]:
      0      1      2
```

(continues on next page)



(continued from previous page)

```

0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

```

```
In [315]: df.to_clipboard()
```

```
In [316]: pd.read_clipboard()
```

```

Out[316]:
      0      1      2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

```

We can see that we got the same content back, which we had earlier written to the clipboard.

---

**Note:** You may need to install `xclip` or `xsel` (with `gtk`, `PyQt5`, `PyQt4` or `qtpy`) on Linux to use these methods.

---

## 24.6 Pickling

All pandas objects are equipped with `to_pickle` methods which use Python's `cPickle` module to save data structures to disk using the pickle format.

```
In [317]: df
```

```

Out[317]:
      0      1      2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

```

```
In [318]: df.to_pickle('foo.pkl')
```

The `read_pickle` function in the pandas namespace can be used to load any pickled pandas object (or any other pickled object) from file:

```
In [319]: pd.read_pickle('foo.pkl')
```

```

Out[319]:
      0      1      2
0 -0.288267 -0.084905  0.004772
1  1.382989  0.343635 -1.253994
2 -0.124925  0.212244  0.496654
3  0.525417  1.238640 -1.210543
4 -1.175743 -0.172372 -0.734129

```

**Warning:** Loading pickled data received from untrusted sources can be unsafe.

See: <https://docs.python.org/3/library/pickle.html>

**Warning:** Several internal refactorings have been done while still preserving compatibility with pickles created with older versions of pandas. However, for such cases, pickled DataFrames, Series etc, must be read with `pd.read_pickle`, rather than `pickle.load`.

See [here](#) and [here](#) for some examples of compatibility-breaking changes. See [this question](#) for a detailed explanation.

### 24.6.1 Compressed pickle files

New in version 0.20.0.

`read_pickle()`, `DataFrame.to_pickle()` and `Series.to_pickle()` can read and write compressed pickle files. The compression types of `gzip`, `bz2`, `xz` are supported for reading and writing. The `zip` file format only supports reading and must contain only one data file to be read.

The compression type can be an explicit parameter or be inferred from the file extension. If ‘infer’, then use `gzip`, `bz2`, `zip`, or `xz` if filename ends in `'.gz'`, `'.bz2'`, `'.zip'`, or `'.xz'`, respectively.

```
In [320]: df = pd.DataFrame({
.....:     'A': np.random.randn(1000),
.....:     'B': 'foo',
.....:     'C': pd.date_range('20130101', periods=1000, freq='s')})
.....:

In [321]: df
Out[321]:
```

	A	B	C
0	0.478412	foo	2013-01-01 00:00:00
1	-0.783748	foo	2013-01-01 00:00:01
2	1.403558	foo	2013-01-01 00:00:02
3	-0.539282	foo	2013-01-01 00:00:03
4	-1.651012	foo	2013-01-01 00:00:04
5	0.692072	foo	2013-01-01 00:00:05
6	1.022171	foo	2013-01-01 00:00:06
..	...	...	...
993	-1.613932	foo	2013-01-01 00:16:33
994	1.088104	foo	2013-01-01 00:16:34
995	-0.632963	foo	2013-01-01 00:16:35
996	-0.585314	foo	2013-01-01 00:16:36
997	-0.275038	foo	2013-01-01 00:16:37
998	-0.937512	foo	2013-01-01 00:16:38
999	0.632369	foo	2013-01-01 00:16:39

[1000 rows x 3 columns]

Using an explicit compression type:

```
In [322]: df.to_pickle("data.pkl.compress", compression="gzip")

In [323]: rt = pd.read_pickle("data.pkl.compress", compression="gzip")

In [324]: rt
```

(continues on next page)

(continued from previous page)

```
Out [324]:
```

	A	B	C
0	0.478412	foo	2013-01-01 00:00:00
1	-0.783748	foo	2013-01-01 00:00:01
2	1.403558	foo	2013-01-01 00:00:02
3	-0.539282	foo	2013-01-01 00:00:03
4	-1.651012	foo	2013-01-01 00:00:04
5	0.692072	foo	2013-01-01 00:00:05
6	1.022171	foo	2013-01-01 00:00:06
..	...	...	...
993	-1.613932	foo	2013-01-01 00:16:33
994	1.088104	foo	2013-01-01 00:16:34
995	-0.632963	foo	2013-01-01 00:16:35
996	-0.585314	foo	2013-01-01 00:16:36
997	-0.275038	foo	2013-01-01 00:16:37
998	-0.937512	foo	2013-01-01 00:16:38
999	0.632369	foo	2013-01-01 00:16:39

[1000 rows x 3 columns]

Inferring compression type from the extension:

```
In [325]: df.to_pickle("data.pkl.xz", compression="infer")

In [326]: rt = pd.read_pickle("data.pkl.xz", compression="infer")

In [327]: rt
Out [327]:
```

	A	B	C
0	0.478412	foo	2013-01-01 00:00:00
1	-0.783748	foo	2013-01-01 00:00:01
2	1.403558	foo	2013-01-01 00:00:02
3	-0.539282	foo	2013-01-01 00:00:03
4	-1.651012	foo	2013-01-01 00:00:04
5	0.692072	foo	2013-01-01 00:00:05
6	1.022171	foo	2013-01-01 00:00:06
..	...	...	...
993	-1.613932	foo	2013-01-01 00:16:33
994	1.088104	foo	2013-01-01 00:16:34
995	-0.632963	foo	2013-01-01 00:16:35
996	-0.585314	foo	2013-01-01 00:16:36
997	-0.275038	foo	2013-01-01 00:16:37
998	-0.937512	foo	2013-01-01 00:16:38
999	0.632369	foo	2013-01-01 00:16:39

[1000 rows x 3 columns]

The default is to ‘infer’:

```
In [328]: df.to_pickle("data.pkl.gz")

In [329]: rt = pd.read_pickle("data.pkl.gz")

In [330]: rt
Out [330]:
```

	A	B	C
0	0.478412	foo	2013-01-01 00:00:00

(continues on next page)

(continued from previous page)

```

1   -0.783748   foo 2013-01-01 00:00:01
2    1.403558   foo 2013-01-01 00:00:02
3   -0.539282   foo 2013-01-01 00:00:03
4   -1.651012   foo 2013-01-01 00:00:04
5    0.692072   foo 2013-01-01 00:00:05
6    1.022171   foo 2013-01-01 00:00:06
..      ...      ...      ...
993 -1.613932   foo 2013-01-01 00:16:33
994  1.088104   foo 2013-01-01 00:16:34
995 -0.632963   foo 2013-01-01 00:16:35
996 -0.585314   foo 2013-01-01 00:16:36
997 -0.275038   foo 2013-01-01 00:16:37
998 -0.937512   foo 2013-01-01 00:16:38
999  0.632369   foo 2013-01-01 00:16:39

```

```
[1000 rows x 3 columns]
```

```
In [331]: df["A"].to_pickle("s1.pkl.bz2")
```

```
In [332]: rt = pd.read_pickle("s1.pkl.bz2")
```

```
In [333]: rt
```

```
Out[333]:
```

```

0      0.478412
1     -0.783748
2      1.403558
3     -0.539282
4     -1.651012
5      0.692072
6      1.022171
...
993   -1.613932
994    1.088104
995   -0.632963
996   -0.585314
997   -0.275038
998   -0.937512
999    0.632369

```

```
Name: A, Length: 1000, dtype: float64
```

## 24.7 msgpack

pandas supports the msgpack format for object serialization. This is a lightweight portable binary format, similar to binary JSON, that is highly space efficient, and provides good performance both on the writing (serialization), and reading (deserialization).

**Warning:** This is a very new feature of pandas. We intend to provide certain optimizations in the io of the msgpack data. Since this is marked as an EXPERIMENTAL LIBRARY, the storage format may not be stable until a future release.

```
In [334]: df = pd.DataFrame(np.random.rand(5, 2), columns=list('AB'))
```

(continues on next page)

(continued from previous page)

```

In [335]: df.to_msgpack('foo.msg')

In [336]: pd.read_msgpack('foo.msg')
Out[336]:
      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359

In [337]: s = pd.Series(np.random.rand(5), index=pd.date_range('20130101', periods=5))

```

You can pass a list of objects and you will receive them back on deserialization.

```

In [338]: pd.to_msgpack('foo.msg', df, 'foo', np.array([1, 2, 3]), s)

In [339]: pd.read_msgpack('foo.msg')
Out[339]:
[
      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359, 'foo', array([1, 2, 3]), 2013-01-01    0.548134
2013-01-02    0.503447
2013-01-03    0.348438
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64]

```

You can pass `iterator=True` to iterate over the unpacked results:

```

In [340]: for o in pd.read_msgpack('foo.msg', iterator=True):
.....:     print(o)
.....:
      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359
foo
[1 2 3]
2013-01-01    0.548134
2013-01-02    0.503447
2013-01-03    0.348438
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64

```

You can pass `append=True` to the writer to append to an existing pack:

```

In [341]: df.to_msgpack('foo.msg', append=True)

In [342]: pd.read_msgpack('foo.msg')
Out[342]:

```

(continues on next page)

(continued from previous page)

```
[
      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359, 'foo', array([1, 2, 3]), 2013-01-01    0.548134
2013-01-02    0.503447
2013-01-03    0.348438
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64,      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359]
```

Unlike other io methods, `to_msgpack` is available on both a per-object basis, `df.to_msgpack()` and using the top-level `pd.to_msgpack(...)` where you can pack arbitrary collections of Python lists, dicts, scalars, while intermixing pandas objects.

```
In [343]: pd.to_msgpack('foo2.msg', {'dict': [{ 'df': df }, {'string': 'foo'},
.....:                                     {'scalar': 1.}, {'s': s}])
.....:

In [344]: pd.read_msgpack('foo2.msg')
Out[344]:
{'dict': ({'df':      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359},
{'string': 'foo'},
{'scalar': 1.0},
{'s': 2013-01-01    0.548134
2013-01-02    0.503447
2013-01-03    0.348438
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64})})
```

## 24.7.1 Read/Write API

Msgpacks can also be read from and written to strings.

```
In [345]: df.to_msgpack()
Out[345]: b'\x84\xa3typ\xadblock_
↪manager\xa5klass\xa9DataFrame\xa4axes\x92\x86\xa3typ\xa5index\xa5klass\xa5Index\xa4name\xc0\xa5dtype
↪index\xa5klass\xaaRangeIndex\xa4name\xc0\xa5start\x00\xa4stop\x05\xa4step\x01\xa6blocks\x91\x86\xa
↪<\xfd\xd2f\xcf\xdc\xc5?0\x15\xebN\xd9\xd2\xea?,\x9c\x16A\xa2@xe5?\xd8/\xdd\xfd
↪"\xc6\xdc?\x11\x1e\x97\x1b\xcdy\xef?&\x1e<\xee\xd6\xa6\xec?p\xd3;\xb2N\xed\xaa?
↪h\xcb\xb1\xbdB\x8b\xd2?\xaf4\x01r"\xe8\xeb?)G6\xd9\xc9\xd1\xe7?
↪\xa5shape\x92\x02\x05\xa5dtype\xa7float64\xa5klass\xaaFloatBlock\xa8compress\xc0'
```

Furthermore you can concatenate the strings to produce a list of the original objects.

```
In [346]: pd.read_msgpack(df.to_msgpack() + s.to_msgpack())
Out[346]:
[
      A      B
0  0.170801  0.895366
1  0.838238  0.052592
2  0.664140  0.289750
3  0.449593  0.872087
4  0.983618  0.744359, 2013-01-01    0.548134
2013-01-02    0.503447
2013-01-03    0.348438
2013-01-04    0.707267
2013-01-05    0.261656
Freq: D, dtype: float64]
```

## 24.8 HDF5 (PyTables)

`HDFStore` is a dict-like object which reads and writes pandas using the high performance HDF5 format using the excellent [PyTables](#) library. See the [cookbook](#) for some advanced strategies

**Warning:** pandas requires `PyTables >= 3.0.0`. There is a indexing bug in `PyTables < 3.2` which may appear when querying stores using an index. If you see a subset of results being returned, upgrade to `PyTables >= 3.2`. Stores created previously will need to be rewritten using the updated version.

```
In [347]: store = pd.HDFStore('store.h5')

In [348]: print(store)
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Objects can be written to the file just like adding key-value pairs to a dict:

```
In [349]: np.random.seed(1234)

In [350]: index = pd.date_range('1/1/2000', periods=8)

In [351]: s = pd.Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [352]: df = pd.DataFrame(randn(8, 3), index=index,
.....:                      columns=['A', 'B', 'C'])
.....:

In [353]: wp = pd.Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
.....:                      major_axis=pd.date_range('1/1/2000', periods=5),
.....:                      minor_axis=['A', 'B', 'C', 'D'])
.....:

# store.put('s', s) is an equivalent method
In [354]: store['s'] = s

In [355]: store['df'] = df

In [356]: store['wp'] = wp
```

(continues on next page)

(continued from previous page)

```
# the type of stored data
In [357]: store.root.wp._v_attrs.pandas_type
Out[357]: 'wide'

In [358]: store
\\Out[358]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

In a current or later Python session, you can retrieve stored objects:

```
# store.get('df') is an equivalent method
In [359]: store['df']
Out[359]:
```

	A	B	C
2000-01-01	0.887163	0.859588	-0.636524
2000-01-02	0.015696	-2.242685	1.150036
2000-01-03	0.991946	0.953324	-2.021255
2000-01-04	-0.334077	0.002118	0.405453
2000-01-05	0.289092	1.321158	-1.546906
2000-01-06	-0.202646	-0.655969	0.193421
2000-01-07	0.553439	1.318152	-0.469305
2000-01-08	0.675554	-1.817027	-0.183109

```
# dotted (attribute) access provides get as well
In [360]: store.df
\\
↪
```

	A	B	C
2000-01-01	0.887163	0.859588	-0.636524
2000-01-02	0.015696	-2.242685	1.150036
2000-01-03	0.991946	0.953324	-2.021255
2000-01-04	-0.334077	0.002118	0.405453
2000-01-05	0.289092	1.321158	-1.546906
2000-01-06	-0.202646	-0.655969	0.193421
2000-01-07	0.553439	1.318152	-0.469305
2000-01-08	0.675554	-1.817027	-0.183109

Deletion of the object specified by the key:

```
# store.remove('wp') is an equivalent method
In [361]: del store['wp']

In [362]: store
Out[362]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

Closing a Store and using a context manager:

```
In [363]: store.close()

In [364]: store
Out[364]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

(continues on next page)



(continued from previous page)

```

In [365]: store.is_open
\\Out[365]: False

# Working with, and automatically closing the store using a context manager
In [366]: with pd.HDFStore('store.h5') as store:
.....:     store.keys()
.....:

```

### 24.8.1 Read/Write API

HDFStore supports an top-level API using `read_hdf` for reading and `to_hdf` for writing, similar to how `read_csv` and `to_csv` work.

```

In [367]: df_t1 = pd.DataFrame(dict(A=list(range(5)), B=list(range(5))))

In [368]: df_t1.to_hdf('store_t1.h5', 'table', append=True)

In [369]: pd.read_hdf('store_t1.h5', 'table', where=['index>2'])
Out[369]:
   A  B
3  3  3
4  4  4

```

HDFStore will by default not drop rows that are all missing. This behavior can be changed by setting `dropna=True`.

```

In [370]: df_with_missing = pd.DataFrame({'col1': [0, np.nan, 2],
.....:                                     'col2': [1, np.nan, np.nan]})
.....:

In [371]: df_with_missing
Out[371]:
   col1  col2
0    0.0    1.0
1    NaN    NaN
2    2.0    NaN

In [372]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
.....:                             format='table', mode='w')
.....:

In [373]: pd.read_hdf('file.h5', 'df_with_missing')
Out[373]:
   col1  col2
0    0.0    1.0
1    NaN    NaN
2    2.0    NaN

In [374]: df_with_missing.to_hdf('file.h5', 'df_with_missing',
.....:                             format='table', mode='w', dropna=True)
.....:

In [375]: pd.read_hdf('file.h5', 'df_with_missing')
Out[375]:
   col1  col2

```

(continues on next page)

(continued from previous page)

```
0    0.0    1.0
2    2.0    NaN
```

This is also true for the major axis of a Panel:

```
In [376]: matrix = [[np.nan, np.nan, np.nan], [1, np.nan, np.nan]],
.....:             [[np.nan, np.nan, np.nan], [np.nan, 5, 6]],
.....:             [[np.nan, np.nan, np.nan], [np.nan, 3, np.nan]]]
.....:

In [377]: panel_with_major_axis_all_missing=pd.Panel(matrix,
.....:        items=['Item1', 'Item2', 'Item3'],
.....:        major_axis=[1, 2],
.....:        minor_axis=['A', 'B', 'C'])
.....:

In [378]: panel_with_major_axis_all_missing
Out[378]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 2 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 1 to 2
Minor_axis axis: A to C

In [379]: panel_with_major_axis_all_missing.to_hdf('file.h5', 'panel',
.....:        dropna=True,
.....:        format='table',
.....:        mode='w')
.....:

In [380]: reloaded = pd.read_hdf('file.h5', 'panel')

In [381]: reloaded
Out[381]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 1 (major_axis) x 3 (minor_axis)
Items axis: Item1 to Item3
Major_axis axis: 2 to 2
Minor_axis axis: A to C
```

## 24.8.2 Fixed Format

The examples above show storing using `put`, which write the HDF5 to PyTables in a fixed array format, called the *fixed* format. These types of stores are **not** appendable once written (though you can simply remove them and rewrite). Nor are they **queryable**; they must be retrieved in their entirety. They also do not support dataframes with non-unique column names. The *fixed* format stores offer very fast writing and slightly faster reading than *table* stores. This format is specified by default when using `put` or `to_hdf` or by `format='fixed'` or `format='f'`.

**Warning:** A fixed format will raise a `TypeError` if you try to retrieve using a `where`:

```
pd.DataFrame(randn(10, 2)).to_hdf('test_fixed.h5', 'df')

pd.read_hdf('test_fixed.h5', 'df', where='index>5')
TypeError: cannot pass a where specification when reading a fixed format.
this store must be selected in its entirety
```

### 24.8.3 Table Format

HDFStore supports another PyTables format on disk, the table format. Conceptually a table is shaped very much like a DataFrame, with rows and columns. A table may be appended to in the same or other sessions. In addition, delete and query type operations are supported. This format is specified by `format='table'` or `format='t'` to append or put or to\_hdf.

This format can be set as an option as well `pd.set_option('io.hdf.default_format', 'table')` to enable put/append/to\_hdf to by default store in the table format.

```
In [382]: store = pd.HDFStore('store.h5')

In [383]: df1 = df[0:4]

In [384]: df2 = df[4:]

# append data (creates a table automatically)
In [385]: store.append('df', df1)

In [386]: store.append('df', df2)

In [387]: store
Out[387]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# select the entire object
In [388]: store.select('df')
\\Out[388]:
          A          B          C
2000-01-01  0.887163  0.859588 -0.636524
2000-01-02  0.015696 -2.242685  1.150036
2000-01-03  0.991946  0.953324 -2.021255
2000-01-04 -0.334077  0.002118  0.405453
2000-01-05  0.289092  1.321158 -1.546906
2000-01-06 -0.202646 -0.655969  0.193421
2000-01-07  0.553439  1.318152 -0.469305
2000-01-08  0.675554 -1.817027 -0.183109

# the type of stored data
In [389]: store.root.df._v_attrs.pandas_type
\\
↪ 'frame_table'
```

**Note:** You can also create a table by passing `format='table'` or `format='t'` to a put operation.

### 24.8.4 Hierarchical Keys

Keys to a store can be specified as a string. These can be in a hierarchical path-name like format (e.g. `foo/bar/bah`), which will generate a hierarchy of sub-stores (or Groups in PyTables parlance). Keys can be specified with

out the leading '/' and are **always** absolute (e.g. 'foo' refers to '/foo'). Removal operations can remove everything in the sub-store and **below**, so be *careful*.

```
In [390]: store.put('foo/bar/bah', df)

In [391]: store.append('food/orange', df)

In [392]: store.append('food/apple', df)

In [393]: store
Out[393]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

# a list of keys are returned
In [394]: store.keys()
\\Out[394]: ['/df',
↪ '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [395]: store.remove('food')

In [396]: store
Out[396]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

**Warning:** Hierarchical keys cannot be retrieved as dotted (attribute) access as described above for items stored under the root node.

```
In [8]: store.foo.bar.bah
AttributeError: 'HDFStore' object has no attribute 'foo'

# you can directly access the actual PyTables node but using the root node
In [9]: store.root.foo.bar.bah
Out[9]:
/foo/bar/bah (Group) ''
  children := ['block0_items' (Array), 'block0_values' (Array), 'axis0' (Array),
↪ 'axis1' (Array)]
```

Instead, use explicit string based keys:

```
In [397]: store['foo/bar/bah']
Out[397]:
```

	A	B	C
2000-01-01	0.887163	0.859588	-0.636524
2000-01-02	0.015696	-2.242685	1.150036
2000-01-03	0.991946	0.953324	-2.021255
2000-01-04	-0.334077	0.002118	0.405453
2000-01-05	0.289092	1.321158	-1.546906
2000-01-06	-0.202646	-0.655969	0.193421
2000-01-07	0.553439	1.318152	-0.469305
2000-01-08	0.675554	-1.817027	-0.183109



(continued from previous page)

```

description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=1),
    "values_block_1": Float32Col(shape=(1,), dflt=0.0, pos=2),
    "values_block_2": Int64Col(shape=(1,), dflt=0, pos=3),
    "values_block_3": Int64Col(shape=(1,), dflt=0, pos=4),
    "values_block_4": BoolCol(shape=(1,), dflt=False, pos=5),
    "values_block_5": StringCol(itemsize=50, shape=(1,), dflt=b'', pos=6)}
byteorder := 'little'
chunkshape := (689,)
autoindex := True
colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}

```

### 24.8.5.2 Storing Multi-Index DataFrames

Storing multi-index DataFrames as tables is very similar to storing/selecting from homogeneous index DataFrames.

```

In [405]: index = pd.MultiIndex(levels=[['foo', 'bar', 'baz', 'qux'],
.....:                                ['one', 'two', 'three']],
.....:                           labels=[[0, 0, 0, 1, 1, 2, 2, 3, 3, 3],
.....:                                [0, 1, 2, 0, 1, 1, 2, 0, 1, 2]],
.....:                           names=['foo', 'bar'])
.....:

In [406]: df_mi = pd.DataFrame(np.random.randn(10, 3), index=index,
.....:                           columns=['A', 'B', 'C'])
.....:

In [407]: df_mi
Out[407]:
           A         B         C
foo bar
foo one  -0.584718  0.816594 -0.081947
      two  -0.344766  0.528288 -1.068989
      three -0.511881  0.291205  0.566534
bar one   0.503592  0.285296  0.484288
      two   1.363482 -0.781105 -0.468018
baz two   1.224574 -1.281108  0.875476
      three -1.710715 -0.450765  0.749164
qux one  -0.203933 -0.182175  0.680656
      two  -1.818499  0.047072  0.394844
      three -0.248432 -0.617707 -0.682884

In [408]: store.append('df_mi', df_mi)

In [409]: store.select('df_mi')
Out[409]:
           A         B         C
foo bar
foo one  -0.584718  0.816594 -0.081947
      two  -0.344766  0.528288 -1.068989
      three -0.511881  0.291205  0.566534
bar one   0.503592  0.285296  0.484288

```

(continues on next page)

(continued from previous page)

```

    two      1.363482 -0.781105 -0.468018
baz two      1.224574 -1.281108  0.875476
    three -1.710715 -0.450765  0.749164
gux one     -0.203933 -0.182175  0.680656
    two     -1.818499  0.047072  0.394844
    three -0.248432 -0.617707 -0.682884

```

```
# the levels are automatically included as data columns
```

```
In [410]: store.select('df_mi', 'foo=bar')
```

```

=====
→
           A          B          C
foo bar
bar one  0.503592  0.285296  0.484288
      two  1.363482 -0.781105 -0.468018

```

## 24.8.6 Querying

### 24.8.6.1 Querying a Table

`select` and `delete` operations have an optional criterion that can be specified to select/delete only a subset of the data. This allows one to have a very large on-disk table and retrieve only a portion of the data.

A query is specified using the `Term` class under the hood, as a boolean expression.

- `index` and `columns` are supported indexers of a `DataFrames`.
- `major_axis`, `minor_axis`, and `items` are supported indexers of the `Panel`.
- if `data_columns` are specified, these can be used as additional indexers.

Valid comparison operators are:

`=`, `==`, `!=`, `>`, `>=`, `<`, `<=`

Valid boolean expressions are combined with:

- `|` : or
- `&` : and
- `( and )` : for grouping

These rules are similar to how boolean expressions are used in pandas for indexing.

---

#### Note:

- `=` will be automatically expanded to the comparison operator `==`
  - `~` is the not operator, but can only be used in very limited circumstances
  - If a list/tuple of expressions is passed they will be combined via `&`
- 

The following are valid expressions:

- `'index >= date'`
- `"columns = ['A', 'D']"`
- `"columns in ['A', 'D']"`

- `'columns = A'`
- `'columns == A'`
- `" ~(columns = ['A', 'B']) "`
- `'index > df.index[3] & string = "bar"'`
- `'(index > df.index[3] & index <= df.index[6]) | string = "bar"'`
- `"ts >= Timestamp('2012-02-01')"`
- `"major_axis>=20130101"`

The indexers are on the left-hand side of the sub-expression:

`columns,major_axis,ts`

The right-hand side of the sub-expression (after a comparison operator) can be:

- functions that will be evaluated, e.g. `Timestamp('2012-02-01')`
- strings, e.g. `"bar"`
- date-like, e.g. `20130101`, or `"20130101"`
- lists, e.g. `"['A', 'B']"`
- variables that are defined in the local names space, e.g. `date`

---

**Note:** Passing a string to a query by interpolating it into the query expression is not recommended. Simply assign the string of interest to a variable and use that variable in an expression. For example, do this

```
string = "HolyMoly"
store.select('df', 'index == string')
```

instead of this

```
string = "HolyMoly"
store.select('df', 'index == %s' % string)
```

The latter will **not** work and will raise a `SyntaxError`. Note that there's a single quote followed by a double quote in the string variable.

If you *must* interpolate, use the `'%r'` format specifier

```
store.select('df', 'index == %r' % string)
```

which will quote string.

---

Here are some examples:

```
In [411]: dfq = pd.DataFrame(randn(10, 4), columns=list('ABCD'),
.....:                      index=pd.date_range('20130101', periods=10))
.....:
In [412]: store.append('dfq', dfq, format='table', data_columns=True)
```

Use boolean expressions, with in-line function evaluation.



```
In [413]: store.select('dfq', "index>pd.Timestamp('20130104') & columns=['A', 'B']")
Out[413]:
```

	A	B
2013-01-05	1.210384	0.797435
2013-01-06	-0.850346	1.176812
2013-01-07	0.984188	-0.121728
2013-01-08	0.796595	-0.474021
2013-01-09	-0.804834	-2.123620
2013-01-10	0.334198	0.536784

Use and inline column reference

```
In [414]: store.select('dfq', where="A>0 or C>0")
Out[414]:
```

	A	B	C	D
2013-01-01	0.436258	-1.703013	0.393711	-0.479324
2013-01-02	-0.299016	0.694103	0.678630	0.239556
2013-01-03	0.151227	0.816127	1.893534	0.639633
2013-01-04	-0.962029	-2.085266	1.930247	-1.735349
2013-01-05	1.210384	0.797435	-0.379811	0.702562
2013-01-07	0.984188	-0.121728	2.365769	0.496143
2013-01-08	0.796595	-0.474021	-0.056696	1.357797
2013-01-10	0.334198	0.536784	-0.743830	-0.320204

Works with a Panel as well.

```
In [415]: store.append('wp', wp)

In [416]: store
Out[416]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

In [417]: store.select('wp', "major_axis>pd.Timestamp('20000102') & minor_axis=['A',
↪ 'B']")
\\Out[417]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to B
```

The columns keyword can be supplied to select a list of columns to be returned, this is equivalent to passing a 'columns=list\_of\_columns\_to\_filter':

```
In [418]: store.select('df', "columns=['A', 'B']")
Out[418]:
```

	A	B
2000-01-01	0.887163	0.859588
2000-01-02	0.015696	-2.242685
2000-01-03	0.991946	0.953324
2000-01-04	-0.334077	0.002118
2000-01-05	0.289092	1.321158
2000-01-06	-0.202646	-0.655969
2000-01-07	0.553439	1.318152
2000-01-08	0.675554	-1.817027

start and stop parameters can be specified to limit the total search space. These are in terms of the total number

of rows in a table.

```
# this is effectively what the storage of a Panel looks like
In [419]: wp.to_frame()
Out[419]:
```

		Item1	Item2
major	minor		
2000-01-01	A	1.058969	0.215269
	B	-0.397840	0.841009
	C	0.337438	-1.445810
	D	1.047579	-1.401973
2000-01-02	A	1.045938	-0.100918
	B	0.863717	-0.548242
	C	-0.122092	-0.144620
...		...	...
2000-01-04	B	0.036142	0.307969
	C	-2.074978	-0.208499
	D	0.247792	1.033801
2000-01-05	A	-0.897157	-2.400454
	B	-0.136795	2.030604
	C	0.018289	-1.142631
	D	0.755414	0.211883

```
[20 rows x 2 columns]

# limiting the search
In [420]: store.select('wp', "major_axis>20000102 & minor_axis=['A', 'B']",
.....:                  start=0, stop=10)
.....:
////////////////////////////////////
```

↩

```
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 1 (major_axis) x 2 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-03 00:00:00 to 2000-01-03 00:00:00
Minor_axis axis: A to B
```

**Note:** `select` will raise a `ValueError` if the query expression has an unknown variable reference. Usually this means that you are trying to select on a column that is **not** a data column.

`select` will raise a `SyntaxError` if the query expression is not valid.

#### 24.8.6.2 Using timedelta64[ns]

You can store and query using the `timedelta64[ns]` type. Terms can be specified in the format: `<float>(<unit>)`, where `float` may be signed (and fractional), and `unit` can be `D`, `s`, `ms`, `us`, `ns` for the `timedelta`. Here's an example:

```
In [421]: from datetime import timedelta

In [422]: dftd = pd.DataFrame(dict(A = pd.Timestamp('20130101'), B = [ pd.Timestamp(
↳ '20130101') + timedelta(days=i, seconds=10) for i in range(10) ]))

In [423]: dftd['C'] = dftd['A'] - dftd['B']
```

(continues on next page)

(continued from previous page)

**In [424]:** dftd**Out [424]:**

	A	B	C
0	2013-01-01	2013-01-01 00:00:10	-1 days +23:59:50
1	2013-01-01	2013-01-02 00:00:10	-2 days +23:59:50
2	2013-01-01	2013-01-03 00:00:10	-3 days +23:59:50
3	2013-01-01	2013-01-04 00:00:10	-4 days +23:59:50
4	2013-01-01	2013-01-05 00:00:10	-5 days +23:59:50
5	2013-01-01	2013-01-06 00:00:10	-6 days +23:59:50
6	2013-01-01	2013-01-07 00:00:10	-7 days +23:59:50
7	2013-01-01	2013-01-08 00:00:10	-8 days +23:59:50
8	2013-01-01	2013-01-09 00:00:10	-9 days +23:59:50
9	2013-01-01	2013-01-10 00:00:10	-10 days +23:59:50

**In [425]:** store.append('dftd', dftd, data\_columns=True)**In [426]:** store.select('dftd', "C<'-3.5D'")**Out [426]:**

	A	B	C
4	2013-01-01	2013-01-05 00:00:10	-5 days +23:59:50
5	2013-01-01	2013-01-06 00:00:10	-6 days +23:59:50
6	2013-01-01	2013-01-07 00:00:10	-7 days +23:59:50
7	2013-01-01	2013-01-08 00:00:10	-8 days +23:59:50
8	2013-01-01	2013-01-09 00:00:10	-9 days +23:59:50
9	2013-01-01	2013-01-10 00:00:10	-10 days +23:59:50

### 24.8.6.3 Indexing

You can create/modify an index for a table with `create_table_index` after data is already in the table (after and append/put operation). Creating a table index is **highly** encouraged. This will speed your queries a great deal when you use a select with the indexed dimension as the where.

**Note:** Indexes are automatically created on the indexables and any data columns you specify. This behavior can be turned off by passing `index=False` to append.

# we have automatically already created an index (in the first section)

**In [427]:** i = store.root.df.table.cols.index.index**In [428]:** i.optlevel, i.kind**Out [428]:** (6, 'medium')

# change an index by passing new parameters

**In [429]:** store.create\_table\_index('df', optlevel=9, kind='full')**In [430]:** i = store.root.df.table.cols.index.index**In [431]:** i.optlevel, i.kind**Out [431]:** (9, 'full')

Oftentimes when appending large amounts of data to a store, it is useful to turn off index creation for each append, then recreate at the end.

```

In [432]: df_1 = pd.DataFrame(randn(10, 2), columns=list('AB'))

In [433]: df_2 = pd.DataFrame(randn(10, 2), columns=list('AB'))

In [434]: st = pd.HDFStore('appends.h5', mode='w')

In [435]: st.append('df', df_1, data_columns=['B'], index=False)

In [436]: st.append('df', df_2, data_columns=['B'], index=False)

In [437]: st.get_storer('df').table
Out[437]:
/df/table (Table(20,)) ''
description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
byteorder := 'little'
chunkshape := (2730,)

```

Then create the index when finished appending.

```

In [438]: st.create_table_index('df', columns=['B'], optlevel=9, kind='full')

In [439]: st.get_storer('df').table
Out[439]:
/df/table (Table(20,)) ''
description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2)}
byteorder := 'little'
chunkshape := (2730,)
autoindex := True
colindexes := {
  "B": Index(9, full, shuffle, zlib(1)).is_csi=True}

In [440]: st.close()

```

See [here](#) for how to create a completely-sorted-index (CSI) on an existing store.

#### 24.8.6.4 Query via Data Columns

You can designate (and index) certain columns that you want to be able to perform queries (other than the *indexable* columns, which you can always query). For instance say you want to perform this common operation, on-disk, and return just the frame that matches this query. You can specify `data_columns = True` to force all columns to be `data_columns`.

```

In [441]: df_dc = df.copy()

In [442]: df_dc['string'] = 'foo'

In [443]: df_dc.loc[df_dc.index[4: 6], 'string'] = np.nan

In [444]: df_dc.loc[df_dc.index[7: 9], 'string'] = 'bar'

```

(continues on next page)

(continued from previous page)

```

In [445]: df_dc['string2'] = 'cool'

In [446]: df_dc.loc[df_dc.index[1: 3], ['B', 'C']] = 1.0

In [447]: df_dc
Out[447]:
           A           B           C string string2
2000-01-01  0.887163  0.859588 -0.636524    foo    cool
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool
2000-01-05  0.289092  1.321158 -1.546906   NaN    cool
2000-01-06 -0.202646 -0.655969  0.193421   NaN    cool
2000-01-07  0.553439  1.318152 -0.469305    foo    cool
2000-01-08  0.675554 -1.817027 -0.183109    bar    cool

# on-disk operations
In [448]: store.append('df_dc', df_dc, data_columns = ['B', 'C', 'string', 'string2'])

In [449]: store.select('df_dc', where='B > 0')
Out[449]:
           A           B           C string string2
2000-01-01  0.887163  0.859588 -0.636524    foo    cool
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool
2000-01-05  0.289092  1.321158 -1.546906   NaN    cool
2000-01-07  0.553439  1.318152 -0.469305    foo    cool

# getting creative
In [450]: store.select('df_dc', 'B > 0 & C > 0 & string == foo')
\////////////////////////////////////////////////////////////////////////////////////////////////////
↩
           A           B           C string string2
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool

# this is in-memory version of this type of selection
In [451]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
\////////////////////////////////////////////////////////////////////////////////////////////////////
↩
           A           B           C string string2
2000-01-02  0.015696  1.000000  1.000000    foo    cool
2000-01-03  0.991946  1.000000  1.000000    foo    cool
2000-01-04 -0.334077  0.002118  0.405453    foo    cool

# we have automagically created this index and the B/C/string/string2
# columns are stored separately as ``PyTables`` columns
In [452]: store.root.df_dc.table
\////////////////////////////////////////////////////////////////////////////////////////////////////
↩
/df_dc/table (Table(8,)) ''
description := {
  "index": Int64Col(shape=(), dflt=0, pos=0),
  "values_block_0": Float64Col(shape=(1,), dflt=0.0, pos=1),
  "B": Float64Col(shape=(), dflt=0.0, pos=2),

```

(continues on next page)

(continued from previous page)

```

"C": Float64Col(shape=(), dflt=0.0, pos=3),
"string": StringCol(itemsizes=3, shape=(), dflt=b'', pos=4),
"string2": StringCol(itemsizes=4, shape=(), dflt=b'', pos=5)}
byteorder := 'little'
chunkshape := (1680,)
autoindex := True
colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "B": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "C": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "string2": Index(6, medium, shuffle, zlib(1)).is_csi=False}

```

There is some performance degradation by making lots of columns into *data columns*, so it is up to the user to designate these. In addition, you cannot change data columns (nor indexables) after the first append/put operation (Of course you can simply read in the data and create a new table!).

### 24.8.6.5 Iterator

You can pass `iterator=True` or `chunksize=number_in_a_chunk` to select and `select_as_multiple` to return an iterator on the results. The default is 50,000 rows returned in a chunk.

```

In [453]: for df in store.select('df', chunksize=3):
.....:     print(df)
.....:

```

	A	B	C
2000-01-01	0.887163	0.859588	-0.636524
2000-01-02	0.015696	-2.242685	1.150036
2000-01-03	0.991946	0.953324	-2.021255
	A	B	C
2000-01-04	-0.334077	0.002118	0.405453
2000-01-05	0.289092	1.321158	-1.546906
2000-01-06	-0.202646	-0.655969	0.193421
	A	B	C
2000-01-07	0.553439	1.318152	-0.469305
2000-01-08	0.675554	-1.817027	-0.183109

**Note:** You can also use the iterator with `read_hdf` which will open, then automatically close the store when finished iterating.

```

for df in pd.read_hdf('store.h5', 'df', chunksize=3):
    print(df)

```

Note, that the `chunksize` keyword applies to the **source** rows. So if you are doing a query, then the `chunksize` will subdivide the total rows in the table and the query applied, returning an iterator on potentially unequal sized chunks.

Here is a recipe for generating a query and using it to create equal sized return chunks.

```

In [454]: dfreq = pd.DataFrame({'number': np.arange(1, 11)})

In [455]: dfreq
Out[455]:
   number

```

(continues on next page)

(continued from previous page)

```

0      1
1      2
2      3
3      4
4      5
5      6
6      7
7      8
8      9
9     10

In [456]: store.append('dfeq', dfeq, data_columns=['number'])

In [457]: def chunks(l, n):
.....:     return [l[i: i+n] for i in range(0, len(l), n)]
.....:

In [458]: evens = [2, 4, 6, 8, 10]

In [459]: coordinates = store.select_as_coordinates('dfeq', 'number=evens')

In [460]: for c in chunks(coordinates, 2):
.....:     print(store.select('dfeq', where=c))
.....:
number
1      2
3      4
number
5      6
7      8
number
9     10

```

### 24.8.6.6 Advanced Queries

#### Select a Single Column

To retrieve a single indexable or data column, use the method `select_column`. This will, for example, enable you to get the index very quickly. These return a `Series` of the result, indexed by the row number. These do not currently accept the `where` selector.

```

In [461]: store.select_column('df_dc', 'index')
Out[461]:
0    2000-01-01
1    2000-01-02
2    2000-01-03
3    2000-01-04
4    2000-01-05
5    2000-01-06
6    2000-01-07
7    2000-01-08
Name: index, dtype: datetime64[ns]

In [462]: store.select_column('df_dc', 'string')

```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
0    foo
1    foo
2    foo
3    foo
4    NaN
5    NaN
6    foo
7    bar
Name: string, dtype: object

```

## Selecting coordinates

Sometimes you want to get the coordinates (a.k.a the index locations) of your query. This returns an `Int64Index` of the resulting locations. These coordinates can also be passed to subsequent `where` operations.

```

In [463]: df_coord = pd.DataFrame(np.random.randn(1000, 2),
.....:                           index=pd.date_range('20000101', periods=1000))
.....:

```

```

In [464]: store.append('df_coord', df_coord)

```

```

In [465]: c = store.select_as_coordinates('df_coord', 'index > 20020101')

```

```

In [466]: c

```

```

Out[466]:
Int64Index([732, 733, 734, 735, 736, 737, 738, 739, 740, 741,
...
          990, 991, 992, 993, 994, 995, 996, 997, 998, 999],
          dtype='int64', length=268)

```

```

In [467]: store.select('df_coord', where=c)

```

```

////////////////////////////////////
↪
              0              1
2002-01-02 -0.178266 -0.064638
2002-01-03 -1.204956 -3.880898
2002-01-04  0.974470  0.415160
2002-01-05  1.751967  0.485011
2002-01-06 -0.170894  0.748870
2002-01-07  0.629793  0.811053
2002-01-08  2.133776  0.238459
...          ...          ...
2002-09-20 -0.181434  0.612399
2002-09-21 -0.763324 -0.354962
2002-09-22 -0.261776  0.812126
2002-09-23  0.482615 -0.886512
2002-09-24 -0.037757 -0.562953
2002-09-25  0.897706  0.383232
2002-09-26 -1.324806  1.139269

[268 rows x 2 columns]

```



## Selecting using a where mask

Sometime your query can involve creating a list of rows to select. Usually this mask would be a resulting index from an indexing operation. This example selects the months of a datetimeindex which are 5.

```
In [468]: df_mask = pd.DataFrame(np.random.randn(1000, 2),
.....:                           index=pd.date_range('20000101', periods=1000))
.....:

In [469]: store.append('df_mask', df_mask)

In [470]: c = store.select_column('df_mask', 'index')

In [471]: where = c[pd.DatetimeIndex(c).month == 5].index

In [472]: store.select('df_mask', where=where)
Out[472]:
```

	0	1
2000-05-01	-1.006245	-0.616759
2000-05-02	0.218940	0.717838
2000-05-03	0.013333	1.348060
2000-05-04	0.662176	-1.050645
2000-05-05	-1.034870	-0.243242
2000-05-06	-0.753366	-1.454329
2000-05-07	-1.022920	-0.476989
...	...	...
2002-05-25	-0.509090	-0.389376
2002-05-26	0.150674	1.164337
2002-05-27	-0.332944	0.115181
2002-05-28	-1.048127	-0.605733
2002-05-29	1.418754	-0.442835
2002-05-30	-0.433200	0.835001
2002-05-31	-1.041278	1.401811

```
[93 rows x 2 columns]
```

## Storer Object

If you want to inspect the stored object, retrieve via `get_storer`. You could use this programmatically to say get the number of rows in an object.

```
In [473]: store.get_storer('df_dc').nrows
Out[473]: 8
```

### 24.8.6.7 Multiple Table Queries

The methods `append_to_multiple` and `select_as_multiple` can perform appending/selecting from multiple tables at once. The idea is to have one table (call it the selector table) that you index most/all of the columns, and perform your queries. The other table(s) are data tables with an index matching the selector table's index. You can then perform a very fast query on the selector table, yet get lots of data back. This method is similar to having a very wide table, but enables more efficient queries.

The `append_to_multiple` method splits a given single DataFrame into multiple tables according to `d`, a dictionary that maps the table names to a list of 'columns' you want in that table. If `None` is used in place of a list, that table will have the remaining unspecified columns of the given DataFrame. The argument `selector` defines which

table is the selector table (which you can make queries from). The argument `dropna` will drop rows from the input `DataFrame` to ensure tables are synchronized. This means that if a row for one of the tables being written to is entirely `np.NaN`, that row will be dropped from all tables.

If `dropna` is `False`, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES**. Remember that entirely `np.NaN` rows are not written to the `HDFStore`, so if you choose to call `dropna=False`, some tables may have more rows than others, and therefore `select_as_multiple` may not work or it may return unexpected results.

```
In [474]: df_mt = pd.DataFrame(randn(8, 6), index=pd.date_range('1/1/2000',
↳ periods=8),
        .....:                                columns=['A', 'B', 'C', 'D', 'E', 'F'])
        .....:
```

```
In [475]: df_mt['foo'] = 'bar'
```

```
In [476]: df_mt.loc[df_mt.index[1], ('A', 'B')] = np.nan
```

```
# you can also create the tables individually
```

```
In [477]: store.append_to_multiple({'df1_mt': ['A', 'B'], 'df2_mt': None },
        .....:                    df_mt, selector='df1_mt')
        .....:
```

```
In [478]: store
```

```
Out[478]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

```
# individual tables were created
```

```
In [479]: store.select('df1_mt')
\\Out[479]:
```

	A	B
2000-01-01	0.714697	0.318215
2000-01-02	NaN	NaN
2000-01-03	-0.086919	0.416905
2000-01-04	0.489131	-0.253340
2000-01-05	-0.382952	-0.397373
2000-01-06	0.538116	0.226388
2000-01-07	-2.073479	-0.115926
2000-01-08	-0.695400	0.402493

```
In [480]: store.select('df2_mt')
```

```
\\
```

```
↳
      C      D      E      F  foo
2000-01-01  0.607460  0.790907  0.852225  0.096696  bar
2000-01-02  0.811031 -0.356817  1.047085  0.664705  bar
2000-01-03 -0.764381 -0.287229 -0.089351 -1.035115  bar
2000-01-04 -1.948100 -0.116556  0.800597 -0.796154  bar
2000-01-05 -0.717627  0.156995 -0.344718 -0.171208  bar
2000-01-06  1.541729  0.205256  1.998065  0.953591  bar
2000-01-07  1.391070  0.303013  1.093347 -0.101000  bar
2000-01-08 -1.507639  0.089575  0.658822 -1.037627  bar
```

```
# as a multiple
```

```
In [481]: store.select_as_multiple(['df1_mt', 'df2_mt'], where=['A>0', 'B>0'],
        .....:                    selector = 'df1_mt')
        .....:
```

(continues on next page)

(continued from previous page)

	A	B	C	D	E	F	foo
2000-01-01	0.714697	0.318215	0.607460	0.790907	0.852225	0.096696	bar
2000-01-06	0.538116	0.226388	1.541729	0.205256	1.998065	0.953591	bar

## 24.8.7 Delete from a Table

You can delete from a table selectively by specifying a `where`. In deleting rows, it is important to understand the `PyTables` deletes rows by erasing the rows, then **moving** the following data. Thus deleting can potentially be a very expensive operation depending on the orientation of your data. This is especially true in higher dimensional objects (`Panel` and `Panel4D`). To get optimal performance, it's worthwhile to have the dimension you are deleting be the first of the indexables.

Data is ordered (on the disk) in terms of the indexables. Here's a simple use case. You store panel-type data, with dates in the `major_axis` and ids in the `minor_axis`. The data is then interleaved like this:

- `date_1 - id_1 - id_2 - . - id_n`
- `date_2 - id_1 - . - id_n`

It should be clear that a delete operation on the `major_axis` will be fairly quick, as one chunk is removed, then the following data moved. On the other hand a delete operation on the `minor_axis` will be very expensive. In this case it would almost certainly be faster to rewrite the table using a `where` that selects all but the missing data.

```
# returns the number of rows deleted
In [482]: store.remove('wp', 'major_axis > 20000102' )
Out[482]: 12

In [483]: store.select('wp')
Out[483]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-02 00:00:00
Minor_axis axis: A to D
```

**Warning:** Please note that **HDF5 DOES NOT RECLAIM SPACE** in the h5 files automatically. Thus, repeatedly deleting (or removing nodes) and adding again, **WILL TEND TO INCREASE THE FILE SIZE**.

To *repack and clean* the file, use `ptrepack`.

## 24.8.8 Notes & Caveats

### 24.8.8.1 Compression

`PyTables` allows the stored data to be compressed. This applies to all kinds of stores, not just tables. Two parameters are used to control compression: `complevel` and `complib`.

**`complevel` specifies if and how hard data is to be compressed.** `complevel=0` and `complevel=None` disables compression and `0 < complevel < 10` enables compression.

`complib` specifies which compression library to use. If nothing is specified the default library `zlib` is used. A compression library usually optimizes for either good compression rates or speed and the results will depend on the type of data. Which type of compression to choose depends on your specific needs and data. The list of supported compression libraries:

- `zlib`: The default compression library. A classic in terms of compression, achieves good compression rates but is somewhat slow.
- `lzo`: Fast compression and decompression.
- `bzip2`: Good compression rates.
- `blosc`: Fast compression and decompression.

New in version 0.20.2: Support for alternative `blosc` compressors:

- `blosc:blosclz`: This is the default compressor for `blosc`
- `blosc:lz4`: A compact, very popular and fast compressor.
- `blosc:lz4hc`: A tweaked version of LZ4, produces better compression ratios at the expense of speed.
- `blosc:snappy`: A popular compressor used in many places.
- `blosc:zlib`: A classic; somewhat slower than the previous ones, but achieving better compression ratios.
- `blosc:zstd`: An extremely well balanced codec; it provides the best compression ratios among the others above, and at reasonably fast speed.

If `complib` is defined as something other than the listed libraries a `ValueError` exception is issued.

---

**Note:** If the library specified with the `complib` option is missing on your platform, compression defaults to `zlib` without further ado.

---

Enable compression for all objects within the file:

```
store_compressed = pd.HDFStore('store_compressed.h5', complevel=9,
                               complib='blosc:blosclz')
```

Or on-the-fly compression (this only applies to tables) in stores where compression is not enabled:

```
store.append('df', df, complib='zlib', complevel=5)
```

### 24.8.8.2 ptrepack

`PyTables` offers better write performance when tables are compressed after they are written, as opposed to turning on compression at the very beginning. You can use the supplied `PyTables` utility `ptrepack`. In addition, `ptrepack` can change compression levels after the fact.

```
ptrepack --chunkshape=auto --propindexes --complevel=9 --complib=blosc in.h5 out.h5
```

Furthermore `ptrepack in.h5 out.h5` will *repack* the file to allow you to reuse previously deleted space. Alternatively, one can simply remove the file and write again, or use the `copy` method.

### 24.8.8.3 Caveats

**Warning:** HDFStore is **not-threadsafe for writing**. The underlying PyTables only supports concurrent reads (via threading or processes). If you need reading and writing *at the same time*, you need to serialize these operations in a single thread in a single process. You will corrupt your data otherwise. See the (GH2397) for more information.

- If you use locks to manage write access between multiple processes, you may want to use `fsync()` before releasing write locks. For convenience you can use `store.flush(fsync=True)` to do this for you.
- Once a table is created its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended
- Be aware that timezones (e.g., `pytz.timezone('US/Eastern')`) are not necessarily equal across time-zone versions. So if data is localized to a specific timezone in the HDFStore using one version of a timezone library and that data is updated with another version, the data will be converted to UTC since these timezones are not considered equal. Either use the same version of timezone library or use `tz_convert` with the updated timezone definition.

**Warning:** PyTables will show a `NaturalNameWarning` if a column name cannot be used as an attribute selector. *Natural* identifiers contain only letters, numbers, and underscores, and may not begin with a number. Other identifiers cannot be used in a `where` clause and are generally a bad idea.

### 24.8.9 DataTypes

HDFStore will map an object dtype to the PyTables underlying dtype. This means the following types are known to work:

Type	Represents missing values
floating: float64, float32, float16	np.nan
integer: int64, int32, int8, uint64, uint32, uint8	
boolean	
datetime64[ns]	NaT
timedelta64[ns]	NaT
categorical: see the section below	
object: strings	np.nan

unicode columns are not supported, and **WILL FAIL**.

#### 24.8.9.1 Categorical Data

You can write data that contains `category` dtypes to a HDFStore. Queries work the same as if it was an object array. However, the `category` typed data is stored in a more efficient manner.

```
In [484]: dfcat = pd.DataFrame({'A': pd.Series(list('aabbcdba')).astype('category'),
.....:                        'B': np.random.randn(8) })
.....:
In [485]: dfcat
```

(continues on next page)

(continued from previous page)

```

Out[485]:
   A      B
0  a  0.603273
1  a  0.262554
2  b -0.979586
3  b  2.132387
4  c  0.892485
5  d  1.996474
6  b  0.231425
7  a  0.980070

In [486]: dfcat.dtypes
////////////////////////////////////Out[486]:
↪
A      category
B      float64
dtype: object

In [487]: cstore = pd.HDFStore('cats.h5', mode='w')

In [488]: cstore.append('dfcat', dfcat, format='table', data_columns=['A'])

In [489]: result = cstore.select('dfcat', where="A in ['b', 'c']")

In [490]: result
Out[490]:
   A      B
2  b -0.979586
3  b  2.132387
4  c  0.892485
6  b  0.231425

In [491]: result.dtypes
////////////////////////////////////Out[491]:
↪
A      category
B      float64
dtype: object

```

### 24.8.9.2 String Columns

#### min\_itemsize

The underlying implementation of `HDFStore` uses a fixed column width (`itemsizes`) for string columns. A string column `itemsize` is calculated as the maximum of the length of data (for that column) that is passed to the `HDFStore`, **in the first append**. Subsequent appends, may introduce a string for a column **larger** than the column can hold, an `Exception` will be raised (otherwise you could have a silent truncation of these columns, leading to loss of information). In the future we may relax this and allow a user-specified truncation to occur.

Pass `min_itemsize` on the first table creation to a-priori specify the minimum length of a particular string column. `min_itemsize` can be an integer, or a dict mapping a column name to an integer. You can pass `values` as a key to allow all *indexables* or *data\_columns* to have this `min_itemsize`.

Passing a `min_itemsize` dict will cause all passed columns to be created as *data\_columns* automatically.

**Note:** If you are not passing any `data_columns`, then the `min_itemsize` will be the maximum of the length of

any string passed

```
In [492]: dfs = pd.DataFrame(dict(A='foo', B='bar'), index=list(range(5)))

In [493]: dfs
Out[493]:
   A  B
0  foo bar
1  foo bar
2  foo bar
3  foo bar
4  foo bar

# A and B have a size of 30
In [494]: store.append('dfs', dfs, min_itemsize=30)

In [495]: store.get_storer('dfs').table
Out[495]:
/dfs/table (Table(5,)) ''
  description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": StringCol(itemsize=30, shape=(2,), dflt=b'', pos=1)}
  byteorder := 'little'
  chunkshape := (963,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False}

# A is created as a data_column with a size of 30
# B is size is calculated
In [496]: store.append('dfs2', dfs, min_itemsize={'A': 30})

In [497]: store.get_storer('dfs2').table
Out[497]:
/dfs2/table (Table(5,)) ''
  description := {
    "index": Int64Col(shape=(), dflt=0, pos=0),
    "values_block_0": StringCol(itemsize=3, shape=(1,), dflt=b'', pos=1),
    "A": StringCol(itemsize=30, shape=(), dflt=b'', pos=2)}
  byteorder := 'little'
  chunkshape := (1598,)
  autoindex := True
  colindexes := {
    "index": Index(6, medium, shuffle, zlib(1)).is_csi=False,
    "A": Index(6, medium, shuffle, zlib(1)).is_csi=False}
```

### nan\_rep

String columns will serialize a `np.nan` (a missing value) with the `nan_rep` string representation. This defaults to the string value `nan`. You could inadvertently turn an actual `nan` value into a missing value.

```
In [498]: dfss = pd.DataFrame(dict(A=['foo', 'bar', 'nan']))

In [499]: dfss
Out[499]:
   A
0  foo
```

(continues on next page)

(continued from previous page)

```

1  bar
2  nan

In [500]: store.append('dfss', dfss)

In [501]: store.select('dfss')
Out[501]:
      A
0  foo
1  bar
2  NaN

# here you need to specify a different nan rep
In [502]: store.append('dfss2', dfss, nan_rep='_nan_')

In [503]: store.select('dfss2')
Out[503]:
      A
0  foo
1  bar
2  nan

```

## 24.8.10 External Compatibility

HDFStore writes table format objects in specific formats suitable for producing loss-less round trips to pandas objects. For external compatibility, HDFStore can read native PyTables format tables.

It is possible to write an HDFStore object that can easily be imported into R using the rhdf5 library ([Package website](#)). Create a table format store like this:

```

In [504]: np.random.seed(1)

In [505]: df_for_r = pd.DataFrame({"first": np.random.rand(100),
.....:                             "second": np.random.rand(100),
.....:                             "class": np.random.randint(0, 2, (100, ))},
.....:                             index=range(100))

In [506]: df_for_r.head()
Out[506]:
      first    second  class
0  0.417022  0.326645     0
1  0.720324  0.527058     0
2  0.000114  0.885942     1
3  0.302333  0.357270     1
4  0.146756  0.908535     1

In [507]: store_export = pd.HDFStore('export.h5')

In [508]: store_export.append('df_for_r', df_for_r, data_columns=df_dc.columns)

In [509]: store_export
Out[509]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5

```



In R this file can be read into a `data.frame` object using the `rhdf5` library. The following example function reads the corresponding column names and data values from the values and assembles them into a `data.frame`:

```
# Load values and column names for all datasets from corresponding nodes and
# insert them into one data.frame object.

library(rhdf5)

loadhdf5data <- function(h5File) {

  listing <- h5ls(h5File)
  # Find all data nodes, values are stored in *_values and corresponding column
  # titles in *_items
  data_nodes <- grep("_values", listing$name)
  name_nodes <- grep("_items", listing$name)
  data_paths = paste(listing$group[data_nodes], listing$name[data_nodes], sep = "/")
  name_paths = paste(listing$group[name_nodes], listing$name[name_nodes], sep = "/")
  columns = list()
  for (idx in seq(data_paths)) {
    # NOTE: matrices returned by h5read have to be transposed to obtain
    # required Fortran order!
    data <- data.frame(t(h5read(h5File, data_paths[idx])))
    names <- t(h5read(h5File, name_paths[idx]))
    entry <- data.frame(data)
    colnames(entry) <- names
    columns <- append(columns, entry)
  }

  data <- data.frame(columns)

  return(data)
}
```

Now you can import the `DataFrame` into R:

```
> data = loadhdf5data("transfer.hdf5")
> head(data)
   first    second class
1 0.4170220047 0.3266449    0
2 0.7203244934 0.5270581    0
3 0.0001143748 0.8859421    1
4 0.3023325726 0.3572698    1
5 0.1467558908 0.9085352    1
6 0.0923385948 0.6233601    1
```

**Note:** The R function lists the entire HDF5 file's contents and assembles the `data.frame` object from all matching nodes, so use this only as a starting point if you have stored multiple `DataFrame` objects to a single HDF5 file.

### 24.8.11 Performance

- `tables` format come with a writing performance penalty as compared to `fixed` stores. The benefit is the ability to append/delete and query (potentially very large amounts of data). Write times are generally longer as compared with regular stores. Query times can be quite fast, especially on an indexed axis.
- You can pass `chunksize=<int>` to `append`, specifying the write chunksize (default is 50000). This will

significantly lower your memory usage on writing.

- You can pass `expectedrows=<int>` to the first `append`, to set the TOTAL number of expected rows that `PyTables` will expect. This will optimize read/write performance.
- Duplicate rows can be written to tables, but are filtered out in selection (with the last items being selected; thus a table is unique on major, minor pairs)
- A `PerformanceWarning` will be raised if you are attempting to store types that will be pickled by `PyTables` (rather than stored as endemic types). See [Here](#) for more information and some solutions.

## 24.9 Feather

New in version 0.20.0.

Feather provides binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy.

Feather is designed to faithfully serialize and de-serialize `DataFrames`, supporting all of the pandas dtypes, including extension dtypes such as categorical and datetime with tz.

Several caveats.

- This is a newer library, and the format, though stable, is not guaranteed to be backward compatible to the earlier versions.
- The format will NOT write an `Index`, or `MultiIndex` for the `DataFrame` and will raise an error if a non-default one is provided. You can `.reset_index()` to store the index or `.reset_index(drop=True)` to ignore it.
- Duplicate column names and non-string columns names are not supported
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

See the [Full Documentation](#).

```
In [510]: df = pd.DataFrame({'a': list('abc'),
.....:                      'b': list(range(1, 4)),
.....:                      'c': np.arange(3, 6).astype('u1'),
.....:                      'd': np.arange(4.0, 7.0, dtype='float64'),
.....:                      'e': [True, False, True],
.....:                      'f': pd.Categorical(list('abc')),
.....:                      'g': pd.date_range('20130101', periods=3),
.....:                      'h': pd.date_range('20130101', periods=3, tz='US/Eastern
↪'),
.....:                      'i': pd.date_range('20130101', periods=3, freq='ns')})

In [511]: df
Out[511]:
```

	a	b	c	d	...	f	g	h
↪					i			
0	a	1	3	4.0	...	a	2013-01-01	2013-01-01 00:00:00-05:00
↪	2013-01-01	00:00:00.000000000						
1	b	2	4	5.0	...	b	2013-01-02	2013-01-02 00:00:00-05:00
↪	2013-01-01	00:00:00.000000001						
2	c	3	5	6.0	...	c	2013-01-03	2013-01-03 00:00:00-05:00
↪	2013-01-01	00:00:00.000000002						

(continues on next page)

(continued from previous page)

[3 rows x 9 columns]

**In [512]:** df.dtypes

```

////////////////////////////////////
↪
a                object
b                int64
c                uint8
d                float64
e                bool
f                category
g                datetime64[ns]
h    datetime64[ns, US/Eastern]
i                datetime64[ns]
dtype: object

```

Write to a feather file.

**In [513]:** df.to\_feather('example.feather')

Read from a feather file.

**In [514]:** result = pd.read\_feather('example.feather')**In [515]:** result**Out[515]:**

```

  a  b  c  d  ...  f  g  h
↪      i
0  a  1  3  4.0  ...  a  2013-01-01  2013-01-01  00:00:00-05:00
↪  2013-01-01  00:00:00.000000000
1  b  2  4  5.0  ...  b  2013-01-02  2013-01-02  00:00:00-05:00
↪  2013-01-01  00:00:00.000000001
2  c  3  5  6.0  ...  c  2013-01-03  2013-01-03  00:00:00-05:00
↪  2013-01-01  00:00:00.000000002

```

[3 rows x 9 columns]

# we preserve dtypes

**In [516]:** result.dtypes

```

////////////////////////////////////
↪
a                object
b                int64
c                uint8
d                float64
e                bool
f                category
g                datetime64[ns]
h    datetime64[ns, US/Eastern]
i                datetime64[ns]
dtype: object

```

## 24.10 Parquet

New in version 0.21.0.

[Apache Parquet](#) provides a partitioned binary columnar serialization for data frames. It is designed to make reading and writing data frames efficient, and to make sharing data across data analysis languages easy. Parquet can use a variety of compression techniques to shrink the file size as much as possible while still maintaining good read performance.

Parquet is designed to faithfully serialize and de-serialize `DataFrame`s, supporting all of the pandas dtypes, including extension dtypes such as datetime with tz.

Several caveats.

- Duplicate column names and non-string columns names are not supported.
- Index level names, if specified, must be strings.
- Categorical dtypes can be serialized to parquet, but will de-serialize as `object` dtype.
- Non supported types include `Period` and actual Python object types. These will raise a helpful error message on an attempt at serialization.

You can specify an `engine` to direct the serialization. This can be one of `pyarrow`, or `fastparquet`, or `auto`. If the engine is NOT specified, then the `pd.options.io.parquet.engine` option is checked; if this is also `auto`, then `pyarrow` is tried, and falling back to `fastparquet`.

See the documentation for [pyarrow](#) and [fastparquet](#).

**Note:** These engines are very similar and should read/write nearly identical parquet format files. Currently `pyarrow` does not support `timedelta` data, `fastparquet`  $\geq 0.1.4$  supports timezone aware datetimes. These libraries differ by having different underlying dependencies (`fastparquet` by using `numba`, while `pyarrow` uses a c-library).

```
In [517]: df = pd.DataFrame({'a': list('abc'),
.....:                      'b': list(range(1, 4)),
.....:                      'c': np.arange(3, 6).astype('u1'),
.....:                      'd': np.arange(4.0, 7.0, dtype='float64'),
.....:                      'e': [True, False, True],
.....:                      'f': pd.date_range('20130101', periods=3),
.....:                      'g': pd.date_range('20130101', periods=3, tz='US/Eastern')})
```

```
In [518]: df
```

```
Out[518]:
```

	a	b	c	d	e	f	g
0	a	1	3	4.0	True	2013-01-01	2013-01-01 00:00:00-05:00
1	b	2	4	5.0	False	2013-01-02	2013-01-02 00:00:00-05:00
2	c	3	5	6.0	True	2013-01-03	2013-01-03 00:00:00-05:00

```
In [519]: df.dtypes
```

```

////////////////////////////////////
a
b
c
d
e
f
object
int64
uint8
float64
bool
datetime64[ns]
```

(continues on next page)

(continued from previous page)

```
g    datetime64[ns, US/Eastern]
dtype: object
```

Write to a parquet file.

```
In [520]: df.to_parquet('example_pa.parquet', engine='pyarrow')
```

```
In [521]: df.to_parquet('example_fp.parquet', engine='fastparquet')
```

Read from a parquet file.

```
In [522]: result = pd.read_parquet('example_fp.parquet', engine='fastparquet')
```

```
In [523]: result = pd.read_parquet('example_pa.parquet', engine='pyarrow')
```

```
In [524]: result.dtypes
```

```
Out [524]:
a                object
b                int64
c                uint8
d                float64
e                 bool
f    datetime64[ns]
g    datetime64[ns, US/Eastern]
dtype: object
```

Read only certain columns of a parquet file.

```
In [525]: result = pd.read_parquet('example_fp.parquet',
.....:                             engine='fastparquet', columns=['a', 'b'])
.....:
```

```
In [526]: result.dtypes
```

```
Out [526]:
a    object
b    int64
dtype: object
```

## 24.11 SQL Queries

The `pandas.io.sql` module provides a collection of query wrappers to both facilitate data retrieval and to reduce dependency on DB-specific API. Database abstraction is provided by SQLAlchemy if installed. In addition you will need a driver library for your database. Examples of such drivers are [psycopg2](#) for PostgreSQL or [pymysql](#) for MySQL. For [SQLite](#) this is included in Python's standard library by default. You can find an overview of supported drivers for each SQL dialect in the [SQLAlchemy docs](#).

If SQLAlchemy is not installed, a fallback is only provided for `sqlite` (and for `mysql` for backwards compatibility, but this is deprecated and will be removed in a future version). This mode requires a Python database adapter which respect the [Python DB-API](#).

See also some [cookbook examples](#) for some advanced strategies.

The key functions are:

<code>read_sql_table(table_name, con[, schema, ...])</code>	Read SQL database table into a DataFrame.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Read SQL query into a DataFrame.
<code>read_sql(sql, con[, index_col, ...])</code>	Read SQL query or database table into a DataFrame.
<code>DataFrame.to_sql(name, con[, schema, ...])</code>	Write records stored in a DataFrame to a SQL database.

### 24.11.1 pandas.read\_sql\_table

`pandas.read_sql_table(table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None)`

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

**Parameters** `table_name` : string

Name of SQL table in database.

`con` : SQLAlchemy connectable (or database string URI)

SQLite DBAPI connection mode not supported.

`schema` : string, default None

Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

`index_col` : string or list of strings, optional, default: None

Column(s) to set as index(MultiIndex).

`coerce_float` : boolean, default True

Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Can result in loss of Precision.

`parse_dates` : list or dict, default: None

- List of column names to parse as dates.
- Dict of {column\_name: format string} where format string is strftime compatible in case of parsing string times or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of {column\_name: arg dict}, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

`columns` : list, default: None

List of column names to select from SQL table

`chunksize` : int, default None

If specified, returns an iterator where `chunksize` is the number of rows to include in each chunk.

**Returns**

**DataFrame**

See also:

`read_sql_query` Read SQL query into a DataFrame.

`read_sql`

## Notes

Any datetime values with time zone information will be converted to UTC.

### 24.11.2 pandas.read\_sql\_query

`pandas.read_sql_query(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, chunksize=None)`

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an `index_col` parameter to use one of the columns as the index, otherwise default integer index will be used.

**Parameters** `sql` : string SQL query or SQLAlchemy Selectable (select or text object)

SQL query to be executed.

`con` : SQLAlchemy connectable(engine/connection), database string URI,

or sqlite3 DBAPI2 connection Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

`index_col` : string or list of strings, optional, default: None

Column(s) to set as index(MultiIndex).

`coerce_float` : boolean, default True

Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point. Useful for SQL result sets.

`params` : list, tuple or dict, optional, default: None

List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported. Eg. for psycopg2, uses %(name)s so use params={'name' : 'value'}

`parse_dates` : list or dict, default: None

- List of column names to parse as dates.
- Dict of {column\_name: format string} where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of {column\_name: arg dict}, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

`chunksize` : int, default None

If specified, return an iterator where `chunksize` is the number of rows to include in each chunk.

## Returns

**DataFrame**

See also:

`read_sql_table` Read SQL database table into a DataFrame.

`read_sql`

## Notes

Any datetime values with time zone information parsed via the `parse_dates` parameter will be converted to UTC.

### 24.11.3 pandas.read\_sql

`pandas.read_sql(sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None)`

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around `read_sql_table` and `read_sql_query` (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to `read_sql_query`, while a database table name will be routed to `read_sql_table`. Note that the delegated function might have more specific notes about their functionality not listed here.

**Parameters** `sql` : string or SQLAlchemy Selectable (select or text object)

SQL query to be executed or a table name.

**con** : SQLAlchemy connectable (engine/connection) or database string URI

or DBAPI2 connection (fallback mode)

Using SQLAlchemy makes it possible to use any DB supported by that library. If a DBAPI2 object, only sqlite3 is supported.

**index\_col** : string or list of strings, optional, default: None

Column(s) to set as index(MultiIndex).

**coerce\_float** : boolean, default True

Attempts to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets.

**params** : list, tuple or dict, optional, default: None

List of parameters to pass to execute method. The syntax used to pass parameters is database driver dependent. Check your database driver documentation for which of the five syntax styles, described in PEP 249's paramstyle, is supported. Eg. for psycopg2, uses `%(name)s` so use `params={'name': 'value'}`

**parse\_dates** : list or dict, default: None

- List of column names to parse as dates.
- Dict of {column\_name: format string} where format string is strftime compatible in case of parsing string times, or is one of (D, s, ns, ms, us) in case of parsing integer timestamps.
- Dict of {column\_name: arg dict}, where the arg dict corresponds to the keyword arguments of `pandas.to_datetime()` Especially useful with databases without native Datetime support, such as SQLite.

**columns** : list, default: None



List of column names to select from SQL table (only used when reading a table).

**chunksize** : int, default None

If specified, return an iterator where *chunksize* is the number of rows to include in each chunk.

#### Returns

**DataFrame**

See also:

[`read\_sql\_table`](#) Read SQL database table into a DataFrame.

[`read\_sql\_query`](#) Read SQL query into a DataFrame.

### 24.11.4 pandas.DataFrame.to\_sql

`DataFrame.to_sql` (*name*, *con*, *schema=None*, *if\_exists='fail'*, *index=True*, *index\_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [\[R16\]](#) are supported. Tables can be newly created, appended to, or overwritten.

**Parameters** **name** : string

Name of SQL table.

**con** : sqlalchemy.engine.Engine or sqlite3.Connection

Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

**schema** : string, optional

Specify the schema (if database flavor supports this). If None, use default schema.

**if\_exists** : { 'fail', 'replace', 'append' }, default 'fail'

How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

**index** : boolean, default True

Write DataFrame index as a column. Uses *index\_label* as the column name in the table.

**index\_label** : string or sequence, default None

Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**chunksize** : int, optional

Rows will be written in batches of this size at a time. By default, all rows will be written at once.

**dtype** : dict, optional

Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

### Raises `ValueError`

When the table already exists and *if\_exists* is 'fail' (the default).

See also:

`pandas.read_sql` read a DataFrame from a table

### References

[R16], [R17]

### Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
   name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just df1.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...           index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
   A
```

(continues on next page)

(continued from previous page)

```
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...          dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

**Note:** The function `read_sql()` is a convenience wrapper around `read_sql_table()` and `read_sql_query()` (and for backward compatibility) and will delegate to specific function depending on the provided input (database table name or sql query). Table names do not need to be quoted if they have special characters.

In the following example, we use the [SQLite](#) SQL database engine. You can use a temporary SQLite database where data are stored in “memory”.

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to. For more information on `create_engine()` and the URI formatting, see the examples below and the SQLAlchemy [documentation](#)

```
In [527]: from sqlalchemy import create_engine

# Create your engine.
In [528]: engine = create_engine('sqlite:///memory:')
```

If you want to manage your own connections you can pass one of those instead:

```
with engine.connect() as conn, conn.begin():
    data = pd.read_sql_table('data', conn)
```

### 24.11.5 Writing DataFrames

Assuming the following data is in a DataFrame `data`, we can insert it into the database using `to_sql()`.

id	Date	Col_1	Col_2	Col_3
26	2012-10-18	X	25.7	True
42	2012-10-19	Y	-12.4	False
63	2012-10-20	Z	5.73	True

```
In [529]: data.to_sql('data', engine)
```

With some databases, writing large DataFrames can result in errors due to packet size limitations being exceeded. This can be avoided by setting the `chunksize` parameter when calling `to_sql`. For example, the following writes data to the database in batches of 1000 rows at a time:

```
In [530]: data.to_sql('data_chunked', engine, chunksize=1000)
```

`to_sql()` will try to map your data to an appropriate SQL data type based on the dtype of the data. When you have columns of dtype object, pandas will try to infer the data type.

You can always override the default type by specifying the desired SQL type of any of the columns by using the `dtype` argument. This argument needs a dictionary mapping column names to SQLAlchemy types (or strings for the sqlite3 fallback mode). For example, specifying to use the sqlalchemy `String` type instead of the default `Text` type for string columns:

```
In [531]: from sqlalchemy.types import String
In [532]: data.to_sql('data_dtype', engine, dtype={'Col_1': String})
```

---

**Note:** Due to the limited support for `timedelta`'s in the different database flavors, columns with type `timedelta64` will be written as integer values as nanoseconds to the database and a warning will be raised.

---

---

**Note:** Columns of `category` dtype will be converted to the dense representation as you would get with `np.asarray(categorical)` (e.g. for string categories this gives an array of strings). Because of this, reading the database table back in does **not** generate a categorical.

---

## 24.11.6 Reading Tables

`read_sql_table()` will read a database table given the table name and optionally a subset of columns to read.

---

**Note:** In order to use `read_sql_table()`, you **must** have the SQLAlchemy optional dependency installed.

---

```
In [533]: pd.read_sql_table('data', engine)
Out[533]:
```

	index	id	Date	Col_1	Col_2	Col_3
0	0	26	2010-10-18	X	27.50	True
1	1	42	2010-10-19	Y	-12.50	False
2	2	63	2010-10-20	Z	5.73	True

You can also specify the name of the column as the `DataFrame` index, and specify a subset of columns to be read.

```
In [534]: pd.read_sql_table('data', engine, index_col='id')
Out[534]:
```

	index	Date	Col_1	Col_2	Col_3
id					
26	0	2010-10-18	X	27.50	True
42	1	2010-10-19	Y	-12.50	False
63	2	2010-10-20	Z	5.73	True

```
In [535]: pd.read_sql_table('data', engine, columns=['Col_1', 'Col_2'])
//////////
↪
Col_1 Col_2
0    X 27.50
1    Y -12.50
2    Z  5.73
```

And you can explicitly force columns to be parsed as dates:

```
In [536]: pd.read_sql_table('data', engine, parse_dates=['Date'])
Out[536]:
```

	index	id	Date	Col_1	Col_2	Col_3
0	0	26	2010-10-18	X	27.50	True
1	1	42	2010-10-19	Y	-12.50	False
2	2	63	2010-10-20	Z	5.73	True

If needed you can explicitly specify a format string, or a dict of arguments to pass to `pandas.to_datetime()`:

```
pd.read_sql_table('data', engine, parse_dates={'Date': '%Y-%m-%d'})
pd.read_sql_table('data', engine, parse_dates={'Date': {'format': '%Y-%m-%d %H:%M:%S'}}
→))
```

You can check if a table exists using `has_table()`

### 24.11.7 Schema support

Reading from and writing to different schema's is supported through the `schema` keyword in the `read_sql_table()` and `to_sql()` functions. Note however that this depends on the database flavor (sqlite does not have schema's). For example:

```
df.to_sql('table', engine, schema='other_schema')
pd.read_sql_table('table', engine, schema='other_schema')
```

### 24.11.8 Querying

You can query using raw SQL in the `read_sql_query()` function. In this case you must use the SQL variant appropriate for your database. When using SQLAlchemy, you can also pass SQLAlchemy Expression language constructs, which are database-agnostic.

```
In [537]: pd.read_sql_query('SELECT * FROM data', engine)
Out[537]:
```

	index	id	Date	Col_1	Col_2	Col_3
0	0	26	2010-10-18 00:00:00.000000	X	27.50	1
1	1	42	2010-10-19 00:00:00.000000	Y	-12.50	0
2	2	63	2010-10-20 00:00:00.000000	Z	5.73	1

Of course, you can specify a more “complex” query.

```
In [538]: pd.read_sql_query("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;",
→engine)
Out[538]:
```

	id	Col_1	Col_2
0	42	Y	-12.5

The `read_sql_query()` function supports a `chunksize` argument. Specifying this will return an iterator through chunks of the query result:

```
In [539]: df = pd.DataFrame(np.random.randn(20, 3), columns=list('abc'))

In [540]: df.to_sql('data_chunks', engine, index=False)
```

```
In [541]: for chunk in pd.read_sql_query("SELECT * FROM data_chunks", engine,
↳ chunksize=5):
    .....:     print(chunk)
    .....:
           a          b          c
0  0.280665 -0.073113  1.160339
1  0.369493  1.904659  1.111057
2  0.659050 -1.627438  0.602319
3  0.420282  0.810952  1.044442
4 -0.400878  0.824006 -0.562305
           a          b          c
0  1.954878 -1.331952 -1.760689
1 -1.650721 -0.890556 -1.119115
2  1.956079 -0.326499 -1.342676
3  1.114383 -0.586524 -1.236853
4  0.875839  0.623362 -0.434957
           a          b          c
0  1.407540  0.129102  1.616950
1  0.502741  1.558806  0.109403
2 -1.219744  2.449369 -0.545774
3 -0.198838 -0.700399 -0.203394
4  0.242669  0.201830  0.661020
           a          b          c
0  1.792158 -0.120465 -1.233121
1 -1.182318 -0.665755 -1.674196
2  0.825030 -0.498214 -0.310985
3 -0.001891 -1.396620 -0.861316
4  0.674712  0.618539 -0.443172
```

You can also run a plain query without creating a DataFrame with `execute()`. This is useful for queries that don't return values, such as INSERT. This is functionally equivalent to calling `execute` on the SQLAlchemy engine or db connection object. Again, you must use the SQL syntax variant appropriate for your database.

```
from pandas.io import sql
sql.execute('SELECT * FROM table_name', engine)
sql.execute('INSERT INTO table_name VALUES(?, ?, ?)', engine,
            params=[('id', 1, 12.2, True)])
```

### 24.11.9 Engine connection examples

To connect with SQLAlchemy you use the `create_engine()` function to create an engine object from database URI. You only need to create the engine once per database you are connecting to.

```
from sqlalchemy import create_engine

engine = create_engine('postgresql://scott:tiger@localhost:5432/mydatabase')

engine = create_engine('mysql+mysqldb://scott:tiger@localhost/foo')

engine = create_engine('oracle://scott:tiger@127.0.0.1:1521/sidname')

engine = create_engine('mssql+pyodbc://mydsn')

# sqlite://<nohostname>/<path>
# where <path> is relative:
```

(continues on next page)

(continued from previous page)

```
engine = create_engine('sqlite:///foo.db')

# or absolute, starting with a slash:
engine = create_engine('sqlite:///absolute/path/to/foo.db')
```

For more information see the examples the [SQLAlchemy documentation](#)

### 24.11.10 Advanced SQLAlchemy queries

You can use SQLAlchemy constructs to describe your query.

Use `sqlalchemy.text()` to specify query parameters in a backend-neutral way

```
In [542]: import sqlalchemy as sa

In [543]: pd.read_sql(sa.text('SELECT * FROM data where Col_1=:coll'),
.....:               engine, params={'coll': 'X'})
.....:
Out[543]:
```

	index	id	Date	Col_1	Col_2	Col_3
0	0	26	2010-10-18 00:00:00.000000	X	27.5	1

If you have an SQLAlchemy description of your database you can express where conditions using SQLAlchemy expressions

```
In [544]: metadata = sa.MetaData()

In [545]: data_table = sa.Table('data', metadata,
.....:   sa.Column('index', sa.Integer),
.....:   sa.Column('Date', sa.DateTime),
.....:   sa.Column('Col_1', sa.String),
.....:   sa.Column('Col_2', sa.Float),
.....:   sa.Column('Col_3', sa.Boolean),
.....: )
.....:

In [546]: pd.read_sql(sa.select([data_table]).where(data_table.c.Col_3 == True),
↪engine)
Out[546]:
```

	index	Date	Col_1	Col_2	Col_3
0	0	2010-10-18	X	27.50	True
1	2	2010-10-20	Z	5.73	True

You can combine SQLAlchemy expressions with parameters passed to `read_sql()` using `sqlalchemy.bindparam()`

```
In [547]: import datetime as dt

In [548]: expr = sa.select([data_table]).where(data_table.c.Date > sa.bindparam('date
↪'))

In [549]: pd.read_sql(expr, engine, params={'date': dt.datetime(2010, 10, 18)})
Out[549]:
```

	index	Date	Col_1	Col_2	Col_3
0	1	2010-10-19	Y	-12.50	False
1	2	2010-10-20	Z	5.73	True

### 24.11.11 Sqlite fallback

The use of sqlite is supported without using SQLAlchemy. This mode requires a Python database adapter which respect the [Python DB-API](#).

You can create connections like so:

```
import sqlite3
con = sqlite3.connect(':memory:')
```

And then issue the following queries:

```
data.to_sql('data', cnx)
pd.read_sql_query("SELECT * FROM data", con)
```

## 24.12 Google BigQuery

**Warning:** Starting in 0.20.0, pandas has split off Google BigQuery support into the separate package `pandas-gbq`. You can `pip install pandas-gbq` to get it.

The `pandas-gbq` package provides functionality to read/write from Google BigQuery.

`pandas` integrates with this external package. if `pandas-gbq` is installed, you can use the `pandas` methods `pd.read_gbq` and `DataFrame.to_gbq`, which will call the respective functions from `pandas-gbq`.

Full documentation can be found [here](#).

## 24.13 Stata Format

### 24.13.1 Writing to Stata format

The method `to_stata()` will write a `DataFrame` into a `.dta` file. The format version of this file is always 115 (Stata 12).

```
In [550]: df = pd.DataFrame(randn(10, 2), columns=list('AB'))
In [551]: df.to_stata('stata.dta')
```

*Stata* data files have limited data type support; only strings with 244 or fewer characters, `int8`, `int16`, `int32`, `float32` and `float64` can be stored in `.dta` files. Additionally, *Stata* reserves certain values to represent missing data. Exporting a non-missing value that is outside of the permitted range in *Stata* for a particular data type will retype the variable to the next larger size. For example, `int8` values are restricted to lie between -127 and 100 in *Stata*, and so variables with values above 100 will trigger a conversion to `int16`. `nan` values in floating points data types are stored as the basic missing data type (`.` in *Stata*).

---

**Note:** It is not possible to export missing data values for integer data types.

---

The *Stata* writer gracefully handles other data types including `int64`, `bool`, `uint8`, `uint16`, `uint32` by casting to the smallest supported type that can represent the data. For example, data with a type of `uint8` will be cast to



`int8` if all values are less than 100 (the upper bound for non-missing `int8` data in *Stata*), or, if values are outside of this range, the variable is cast to `int16`.

**Warning:** Conversion from `int64` to `float64` may result in a loss of precision if `int64` values are larger than  $2^{**53}$ .

**Warning:** `StataWriter` and `to_stata()` only support fixed width strings containing up to 244 characters, a limitation imposed by the version 115 dta file format. Attempting to write *Stata* dta files with strings longer than 244 characters raises a `ValueError`.

### 24.13.2 Reading from Stata format

The top-level function `read_stata` will read a dta file and return either a `DataFrame` or a `StataReader` that can be used to read the file incrementally.

```
In [552]: pd.read_stata('stata.dta')
Out[552]:
```

	index	A	B
0	0	1.810535	-1.305727
1	1	-0.344987	-0.230840
2	2	-2.793085	1.937529
3	3	0.366332	-1.044589
4	4	2.051173	0.585662
5	5	0.429526	-0.606998
6	6	0.106223	-1.525680
7	7	0.795026	-0.374438
8	8	0.134048	1.202055
9	9	0.284748	0.262467

Specifying a `chunksize` yields a `StataReader` instance that can be used to read `chunksize` lines from the file at a time. The `StataReader` object can be used as an iterator.

```
In [553]: reader = pd.read_stata('stata.dta', chunksize=3)

In [554]: for df in reader:
.....:     print(df.shape)
.....:
(3, 3)
(3, 3)
(3, 3)
(1, 3)
```

For more fine-grained control, use `iterator=True` and specify `chunksize` with each call to `read()`.

```
In [555]: reader = pd.read_stata('stata.dta', iterator=True)

In [556]: chunk1 = reader.read(5)

In [557]: chunk2 = reader.read(5)
```

Currently the `index` is retrieved as a column.

The parameter `convert_categoricals` indicates whether value labels should be read and used to create a

Categorical variable from them. Value labels can also be retrieved by the function `value_labels`, which requires `read()` to be called before use.

The parameter `convert_missing` indicates whether missing value representations in Stata should be preserved. If `False` (the default), missing values are represented as `np.nan`. If `True`, missing values are represented using `StataMissingValue` objects, and columns containing missing values will have object data type.

---

**Note:** `read_stata()` and `StataReader` support .dta formats 113-115 (Stata 10-12), 117 (Stata 13), and 118 (Stata 14).

---

---

**Note:** Setting `preserve_dtypes=False` will upcast to the standard pandas data types: `int64` for all integer types and `float64` for floating point data. By default, the Stata data types are preserved when importing.

---

### 24.13.2.1 Categorical Data

Categorical data can be exported to *Stata* data files as value labeled data. The exported data consists of the underlying category codes as integer data values and the categories as value labels. *Stata* does not have an explicit equivalent to a `Categorical` and information about *whether* the variable is ordered is lost when exporting.

**Warning:** *Stata* only supports string value labels, and so `str` is called on the categories when exporting data. Exporting `Categorical` variables with non-string categories produces a warning, and can result a loss of information if the `str` representations of the categories are not unique.

Labeled data can similarly be imported from *Stata* data files as `Categorical` variables using the keyword argument `convert_categoricals` (`True` by default). The keyword argument `order_categoricals` (`True` by default) determines whether imported `Categorical` variables are ordered.

---

**Note:** When importing categorical data, the values of the variables in the *Stata* data file are not preserved since `Categorical` variables always use integer data types between `-1` and `n-1` where `n` is the number of categories. If the original values in the *Stata* data file are required, these can be imported by setting `convert_categoricals=False`, which will import original data (but not the variable labels). The original values can be matched to the imported categorical data since there is a simple mapping between the original *Stata* data values and the category codes of imported `Categorical` variables: missing values are assigned code `-1`, and the smallest original value is assigned `0`, the second smallest is assigned `1` and so on until the largest original value is assigned the code `n-1`.

---

---

**Note:** *Stata* supports partially labeled series. These series have value labels for some but not all data values. Importing a partially labeled series will produce a `Categorical` with string categories for the values that are labeled and numeric categories for values with no label.

---

## 24.14 SAS Formats

The top-level function `read_sas()` can read (but not write) SAS *xport* (.XPT) and (since v0.18.0) *SAS7BDAT* (.sas7bdat) format files.

SAS files only contain two value types: ASCII text and floating point values (usually 8 bytes but sometimes truncated). For xport files, there is no automatic type conversion to integers, dates, or categoricals. For SAS7BDAT files, the format codes may allow date variables to be automatically converted to dates. By default the whole file is read and returned as a `DataFrame`.

Specify a `chunksize` or use `iterator=True` to obtain reader objects (`XportReader` or `SAS7BDATReader`) for incrementally reading the file. The reader objects also have attributes that contain additional information about the file and its variables.

Read a SAS7BDAT file:

```
df = pd.read_sas('sas_data.sas7bdat')
```

Obtain an iterator and read an XPORT file 100,000 lines at a time:

```
rdr = pd.read_sas('sas_xport.xpt', chunk=100000)
for chunk in rdr:
    do_something(chunk)
```

The [specification](#) for the xport file format is available from the SAS web site.

No official documentation is available for the SAS7BDAT format.

## 24.15 Other file formats

pandas itself only supports IO with a limited set of file formats that map cleanly to its tabular data model. For reading and writing other file formats into and from pandas, we recommend these packages from the broader community.

### 24.15.1 netCDF

[xarray](#) provides data structures inspired by the pandas `DataFrame` for working with multi-dimensional datasets, with a focus on the netCDF file format and easy conversion to and from pandas.

## 24.16 Performance Considerations

This is an informal comparison of various IO methods, using pandas 0.20.3. Timings are machine dependent and small differences should be ignored.

```
In [1]: sz = 1000000
In [2]: df = pd.DataFrame({'A': randn(sz), 'B': [1] * sz})

In [3]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 2 columns):
A      1000000 non-null float64
B      1000000 non-null int64
dtypes: float64(1), int64(1)
memory usage: 15.3 MB
```

When writing, the top-three functions in terms of speed are `test_pickle_write`, `test_feather_write` and `test_hdf_fixed_write_compress`.

```
In [14]: %timeit test_sql_write(df)
2.37 s ± 36.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [15]: %timeit test_hdf_fixed_write(df)
194 ms ± 65.9 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [26]: %timeit test_hdf_fixed_write_compress(df)
119 ms ± 2.15 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [16]: %timeit test_hdf_table_write(df)
623 ms ± 125 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [27]: %timeit test_hdf_table_write_compress(df)
563 ms ± 23.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [17]: %timeit test_csv_write(df)
3.13 s ± 49.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [30]: %timeit test_feather_write(df)
103 ms ± 5.88 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [31]: %timeit test_pickle_write(df)
109 ms ± 3.72 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [32]: %timeit test_pickle_write_compress(df)
3.33 s ± 55.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

When reading, the top three are `test_feather_read`, `test_pickle_read` and `test_hdf_fixed_read`.

```
In [18]: %timeit test_sql_read()
1.35 s ± 14.7 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [19]: %timeit test_hdf_fixed_read()
14.3 ms ± 438 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [28]: %timeit test_hdf_fixed_read_compress()
23.5 ms ± 672 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [20]: %timeit test_hdf_table_read()
35.4 ms ± 314 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [29]: %timeit test_hdf_table_read_compress()
42.6 ms ± 2.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [22]: %timeit test_csv_read()
516 ms ± 27.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [33]: %timeit test_feather_read()
4.06 ms ± 115 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [34]: %timeit test_pickle_read()
6.5 ms ± 172 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [35]: %timeit test_pickle_read_compress()
588 ms ± 3.57 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Space on disk (in bytes)

```

34816000 Aug 21 18:00 test.sql
24009240 Aug 21 18:00 test_fixed.hdf
 7919610 Aug 21 18:00 test_fixed_compress.hdf
24458892 Aug 21 18:00 test_table.hdf
 8657116 Aug 21 18:00 test_table_compress.hdf
28520770 Aug 21 18:00 test.csv
16000248 Aug 21 18:00 test.feather
16000848 Aug 21 18:00 test.pkl
 7554108 Aug 21 18:00 test.pkl.compress

```

And here's the code:

```

import os
import pandas as pd
import sqlite3
from numpy.random import randn
from pandas.io import sql

sz = 1000000
df = pd.DataFrame({'A': randn(sz), 'B': [1] * sz})

def test_sql_write(df):
    if os.path.exists('test.sql'):
        os.remove('test.sql')
    sql_db = sqlite3.connect('test.sql')
    df.to_sql(name='test_table', con=sql_db)
    sql_db.close()

def test_sql_read():
    sql_db = sqlite3.connect('test.sql')
    pd.read_sql_query("select * from test_table", sql_db)
    sql_db.close()

def test_hdf_fixed_write(df):
    df.to_hdf('test_fixed.hdf', 'test', mode='w')

def test_hdf_fixed_read():
    pd.read_hdf('test_fixed.hdf', 'test')

def test_hdf_fixed_write_compress(df):
    df.to_hdf('test_fixed_compress.hdf', 'test', mode='w', complib='blosc')

def test_hdf_fixed_read_compress():
    pd.read_hdf('test_fixed_compress.hdf', 'test')

def test_hdf_table_write(df):
    df.to_hdf('test_table.hdf', 'test', mode='w', format='table')

def test_hdf_table_read():
    pd.read_hdf('test_table.hdf', 'test')

def test_hdf_table_write_compress(df):
    df.to_hdf('test_table_compress.hdf', 'test', mode='w', complib='blosc', format=
    ↪ 'table')

def test_hdf_table_read_compress():
    pd.read_hdf('test_table_compress.hdf', 'test')

```

(continues on next page)

(continued from previous page)

```
def test_csv_write(df):
    df.to_csv('test.csv', mode='w')

def test_csv_read():
    pd.read_csv('test.csv', index_col=0)

def test_feather_write(df):
    df.to_feather('test.feather')

def test_feather_read():
    pd.read_feather('test.feather')

def test_pickle_write(df):
    df.to_pickle('test.pkl')

def test_pickle_read():
    pd.read_pickle('test.pkl')

def test_pickle_write_compress(df):
    df.to_pickle('test.pkl.compress', compression='xz')

def test_pickle_read_compress():
    pd.read_pickle('test.pkl.compress', compression='xz')
```

## ENHANCING PERFORMANCE

In this part of the tutorial, we will investigate how to speed up certain functions operating on pandas `DataFrames` using three different techniques: Cython, Numba and `pandas.eval()`. We will see a speed improvement of ~200 when we use Cython and Numba on a test function operating row-wise on the `DataFrame`. Using `pandas.eval()` we will speed up a sum by an order of ~2.

### 25.1 Cython (Writing C extensions for pandas)

For many use cases writing pandas in pure Python and NumPy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizeable speed-ups by offloading work to [cython](#).

This tutorial assumes you have refactored as much as possible in Python, for example by trying to remove for-loops and making use of NumPy vectorization. It's always worth optimising in Python first.

This tutorial walks through a “typical” process of cythonizing a slow computation. We use an [example from the Cython documentation](#) but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure Python solution.

#### 25.1.1 Pure python

We have a `DataFrame` to which we want to apply a function row-wise.

```
In [1]: df = pd.DataFrame({'a': np.random.randn(1000),
...:                      'b': np.random.randn(1000),
...:                      'N': np.random.randint(100, 1000, (1000)),
...:                      'x': 'x'})
...:

In [2]: df
Out[2]:
```

	a	b	N	x
0	0.469112	-0.218470	585	x
1	-0.282863	-0.061645	841	x
2	-1.509059	-0.723780	251	x
3	-1.135632	0.551225	972	x
4	1.212112	-0.497767	181	x
5	-0.173215	0.837519	458	x
6	0.119209	1.103245	159	x
..	...	...	...	..
993	0.131892	0.290162	190	x
994	0.342097	0.215341	931	x

(continues on next page)

(continued from previous page)

```

995 -1.512743  0.874737  374  x
996  0.933753  1.120790  246  x
997 -0.308013  0.198768  157  x
998 -0.079915  1.757555  977  x
999 -1.010589 -1.115680  770  x

```

```
[1000 rows x 4 columns]
```

Here's the function in pure Python:

```

In [3]: def f(x):
...:     return x * (x - 1)
...:

In [4]: def integrate_f(a, b, N):
...:     s = 0
...:     dx = (b - a) / N
...:     for i in range(N):
...:         s += f(a + i * dx)
...:     return s * dx
...:

```

We achieve our result by using `apply` (row-wise):

```

In [7]: %timeit df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 174 ms per loop

```

But clearly this isn't fast enough for us. Let's take a look and see where the time is spent during this operation (limited to the most time consuming four calls) using the `prun` `ipython` magic function:

```

In [5]: %prun -l 4 df.apply(lambda x: integrate_f(x['a'], x['b'], x['N']), axis=1)
671713 function calls (666693 primitive calls) in 0.236 seconds

Ordered by: internal time
List reduced from 214 to 4 due to restriction <4>

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
   1000    0.117    0.000    0.173    0.000  <ipython-input-4-c2a74e076cf0>
->:1(integrate_f)
  552423    0.055    0.000    0.055    0.000  <ipython-input-3-c138bdd570e3>:1(f)
    3000    0.008    0.000    0.041    0.000  base.py:3090(get_value)
    3000    0.004    0.000    0.047    0.000  series.py:764(__getitem__)

```

By far the majority of time is spent inside either `integrate_f` or `f`, hence we'll concentrate our efforts cythonizing these two functions.

---

**Note:** In Python 2 replacing the `range` with its generator counterpart (`xrange`) would mean the `range` line would vanish. In Python 3 `range` is already a generator.

---

## 25.1.2 Plain Cython

First we're going to need to import the Cython magic function to `ipython`:



```
In [6]: %load_ext Cython
```

Now, let's simply copy our functions over to Cython as is (the suffix is here to distinguish between function versions):

```
In [7]: %%cython
...: def f_plain(x):
...:     return x * (x - 1)
...: def integrate_f_plain(a, b, N):
...:     s = 0
...:     dx = (b - a) / N
...:     for i in range(N):
...:         s += f_plain(a + i * dx)
...:     return s * dx
...:
```

**Note:** If you're having trouble pasting the above into your ipython, you may need to be using bleeding edge ipython for paste to play well with cell magics.

```
In [4]: %timeit df.apply(lambda x: integrate_f_plain(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 85.5 ms per loop
```

Already this has shaved a third off, not too bad for a simple copy and paste.

### 25.1.3 Adding type

We get another huge improvement simply by providing type information:

```
In [8]: %%cython
...: cdef double f_typed(double x) except? -2:
...:     return x * (x - 1)
...: cpdef double integrate_f_typed(double a, double b, int N):
...:     cdef int i
...:     cdef double s, dx
...:     s = 0
...:     dx = (b - a) / N
...:     for i in range(N):
...:         s += f_typed(a + i * dx)
...:     return s * dx
...:
```

```
In [4]: %timeit df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']), axis=1)
10 loops, best of 3: 20.3 ms per loop
```

Now, we're talking! It's now over ten times faster than the original python implementation, and we haven't *really* modified the code. Let's have another look at what's eating up time:

```
In [9]: %prun -l 4 df.apply(lambda x: integrate_f_typed(x['a'], x['b'], x['N']),
↪axis=1)
119288 function calls (114268 primitive calls) in 0.055 seconds

Ordered by: internal time
List reduced from 211 to 4 due to restriction <4>
```

(continues on next page)

(continued from previous page)

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
3000	0.007	0.000	0.036	0.000	base.py:3090(get_value)
3000	0.004	0.000	0.041	0.000	series.py:764(__getitem__)
9117	0.003	0.000	0.007	0.000	{built-in method builtins.getattr}
1	0.003	0.003	0.053	0.053	{pandas._libs.reduction.reduce}

### 25.1.4 Using ndarray

It's calling series... a lot! It's creating a Series from each row, and get-ting from both the index and the series (three times for each row). Function calls are expensive in Python, so maybe we could minimize these by cythonizing the apply part.

**Note:** We are now passing ndarrays into the Cython function, fortunately Cython plays very nicely with NumPy.

```
In [10]: %%cython
...: cimport numpy as np
...: import numpy as np
...: cdef double f_typed(double x) except? -2:
...:     return x * (x - 1)
...: cpdef double integrate_f_typed(double a, double b, int N):
...:     cdef int i
...:     cdef double s, dx
...:     s = 0
...:     dx = (b - a) / N
...:     for i in range(N):
...:         s += f_typed(a + i * dx)
...:     return s * dx
...: cpdef np.ndarray[double] apply_integrate_f(np.ndarray col_a, np.ndarray col_
↪ b, np.ndarray col_N):
...:     assert (col_a.dtype == np.float and col_b.dtype == np.float and col_N.
↪ dtype == np.int)
...:     cdef Py_ssize_t i, n = len(col_N)
...:     assert (len(col_a) == len(col_b) == n)
...:     cdef np.ndarray[double] res = np.empty(n)
...:     for i in range(len(col_a)):
...:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
...:     return res
...:
```

The implementation is simple, it creates an array of zeros and loops over the rows, applying our `integrate_f_typed`, and putting this in the zeros array.

**Warning:** You can **not** pass a Series directly as a ndarray typed parameter to a Cython function. Instead pass the actual ndarray using the `.values` attribute of the Series. The reason is that the Cython definition is specific to an ndarray and not the passed Series.

So, do not do this:

```
apply_integrate_f(df['a'], df['b'], df['N'])
```

But rather, use `.values` to get the underlying ndarray:

```
apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
```

**Note:** Loops like this would be *extremely* slow in Python, but in Cython looping over NumPy arrays is *fast*.

```
In [4]: %timeit apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
1000 loops, best of 3: 1.25 ms per loop
```

We've gotten another big improvement. Let's check again where the time is spent:

```
In [11]: %prun -l 4 apply_integrate_f(df['a'].values, df['b'].values, df['N'].values)
215 function calls in 0.001 seconds
```

Ordered by: internal time

List reduced from 55 to 4 due to restriction <4>

	ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
	1	0.001	0.001	0.001	0.001	{built-in method _cython_magic_
→	039cb0af339d11610fbd07bd4033a1cf.					apply_integrate_f}
	1	0.000	0.000	0.001	0.001	{built-in method builtins.exec}
	3	0.000	0.000	0.000	0.000	internals.py:4137(iget)
	3	0.000	0.000	0.000	0.000	generic.py:2484(_get_item_cache)

As one might expect, the majority of the time is now spent in `apply_integrate_f`, so if we wanted to make anymore efficiencies we must continue to concentrate our efforts here.

### 25.1.5 More advanced techniques

There is still hope for improvement. Here's an example of using some more advanced Cython techniques:

```
In [12]: %%cython
....: cimport cython
....: cimport numpy as np
....: import numpy as np
....: cdef double f_typed(double x) except? -2:
....:     return x * (x - 1)
....: cpdef double integrate_f_typed(double a, double b, int N):
....:     cdef int i
....:     cdef double s, dx
....:     s = 0
....:     dx = (b - a) / N
....:     for i in range(N):
....:         s += f_typed(a + i * dx)
....:     return s * dx
....: @cython.boundscheck(False)
....: @cython.wraparound(False)
....: cpdef np.ndarray[double] apply_integrate_f_wrap(np.ndarray[double] col_a, np.
→ ndarray[double] col_b, np.ndarray[int] col_N):
....:     cdef int i, n = len(col_N)
....:     assert len(col_a) == len(col_b) == n
....:     cdef np.ndarray[double] res = np.empty(n)
....:     for i in range(n):
....:         res[i] = integrate_f_typed(col_a[i], col_b[i], col_N[i])
....:     return res
....:
```

```
In [4]: %timeit apply_integrate_f_wrap(df['a'].values, df['b'].values, df['N'].values)
1000 loops, best of 3: 987 us per loop
```

Even faster, with the caveat that a bug in our Cython code (an off-by-one error, for example) might cause a segfault because memory access isn't checked. For more about boundscheck and wraparound, see the Cython docs on [compiler directives](#).

## 25.2 Using Numba

A recent alternative to statically compiling Cython code, is to use a *dynamic jit-compiler*, Numba.

Numba gives you the power to speed up your applications with high performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to run on either CPU or GPU hardware, and is designed to integrate with the Python scientific software stack.

---

**Note:** You will need to install Numba. This is easy with conda, by using: `conda install numba`, see [installing using miniconda](#).

---

---

**Note:** As of Numba version 0.20, pandas objects cannot be passed directly to Numba-compiled functions. Instead, one must pass the NumPy array underlying the pandas object to the Numba-compiled function as demonstrated below.

---

### 25.2.1 Jit

We demonstrate how to use Numba to just-in-time compile our code. We simply take the plain Python code from above and annotate with the `@jit` decorator.

```
import numba

@numba.jit
def f_plain(x):
    return x * (x - 1)

@numba.jit
def integrate_f_numba(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_plain(a + i * dx)
    return s * dx

@numba.jit
def apply_integrate_f_numba(col_a, col_b, col_N):
    n = len(col_N)
    result = np.empty(n, dtype='float64')
    assert len(col_a) == len(col_b) == n
    for i in range(n):
```

(continues on next page)

(continued from previous page)

```

        result[i] = integrate_f_numba(col_a[i], col_b[i], col_N[i])
    return result

def compute_numba(df):
    result = apply_integrate_f_numba(df['a'].values, df['b'].values, df['N'].values)
    return pd.Series(result, index=df.index, name='result')
```

Note that we directly pass NumPy arrays to the Numba function. `compute_numba` is just a wrapper that provides a nicer interface by passing/returning pandas objects.

```

In [4]: %timeit compute_numba(df)
1000 loops, best of 3: 798 us per loop
```

In this example, using Numba was faster than Cython.

## 25.2.2 Vectorize

Numba can also be used to write vectorized functions that do not require the user to explicitly loop over the observations of a vector; a vectorized function will be applied to each row automatically. Consider the following toy example of doubling each observation:

```

import numba

def double_every_value_nonumba(x):
    return x*2

@numba.vectorize
def double_every_value_withnumba(x):
    return x*2

# Custom function without numba
In [5]: %timeit df['coll_doubled'] = df.a.apply(double_every_value_nonumba)
1000 loops, best of 3: 797 us per loop

# Standard implementation (faster than a custom function)
In [6]: %timeit df['coll_doubled'] = df.a*2
1000 loops, best of 3: 233 us per loop

# Custom function with numba
In [7]: %timeit df['coll_doubled'] = double_every_value_withnumba(df.a.values)
1000 loops, best of 3: 145 us per loop
```

## 25.2.3 Caveats

---

**Note:** Numba will execute on any function, but can only accelerate certain classes of functions.

---

Numba is best at accelerating functions that apply numerical functions to NumPy arrays. When passed a function that only uses operations it knows how to accelerate, it will execute in `nopython` mode.

If Numba is passed a function that includes something it doesn't know how to work with – a category that currently includes sets, lists, dictionaries, or string functions – it will revert to `object` mode. In `object` mode, Numba

will execute but your code will not speed up significantly. If you would prefer that Numba throw an error if it cannot compile a function in a way that speeds up your code, pass Numba the argument `nopython=True` (e.g. `@numba.jit(nopython=True)`). For more on troubleshooting Numba modes, see the [Numba troubleshooting page](#).

Read more in the [Numba docs](#).

## 25.3 Expression Evaluation via `eval()`

The top-level function `pandas.eval()` implements expression evaluation of *Series* and *DataFrame* objects.

---

**Note:** To benefit from using `eval()` you need to install `numexpr`. See the [recommended dependencies section](#) for more details.

---

The point of using `eval()` for expression evaluation rather than plain Python is two-fold: 1) large *DataFrame* objects are evaluated more efficiently and 2) large arithmetic and boolean expressions are evaluated all at once by the underlying engine (by default `numexpr` is used for evaluation).

---

**Note:** You should not use `eval()` for simple expressions or for expressions involving small *DataFrames*. In fact, `eval()` is many orders of magnitude slower for smaller expressions/objects than plain ol' Python. A good rule of thumb is to only use `eval()` when you have a *DataFrame* with more than 10,000 rows.

---

`eval()` supports all arithmetic expressions supported by the engine in addition to some extensions available only in pandas.

---

**Note:** The larger the frame and the larger the expression the more speedup you will see from using `eval()`.

---

### 25.3.1 Supported Syntax

These operations are supported by `pandas.eval()`:

- Arithmetic operations except for the left shift (`<<`) and right shift (`>>`) operators, e.g., `df + 2 * pi / s ** 4 % 42 - the_golden_ratio`
- Comparison operations, including chained comparisons, e.g., `2 < df < df2`
- Boolean operations, e.g., `df < df2 and df3 < df4 or not df_bool`
- list and tuple literals, e.g., `[1, 2]` or `(1, 2)`
- Attribute access, e.g., `df.a`
- Subscript expressions, e.g., `df[0]`
- Simple variable evaluation, e.g., `pd.eval('df')` (this is not very useful)
- Math functions: `sin`, `cos`, `exp`, `log`, `expm1`, `log1p`, `sqrt`, `sinh`, `cosh`, `tanh`, `arcsin`, `arccos`, `arctan`, `arccosh`, `arcsinh`, `arctanh`, `abs` and `arctan2`.

This Python syntax is **not** allowed:

- Expressions
  - Function calls other than math functions.

- is/is not operations
- if expressions
- lambda expressions
- list/set/dict comprehensions
- Literal dict and set expressions
- yield expressions
- Generator expressions
- Boolean expressions consisting of only scalar values
- Statements
  - Neither `simple` nor `compound` statements are allowed. This includes things like `for`, `while`, and `if`.

### 25.3.2 `eval()` Examples

`pandas.eval()` works well with expressions containing large arrays.

First let's create a few decent-sized arrays to play with:

```
In [13]: nrows, ncols = 20000, 100

In [14]: df1, df2, df3, df4 = [pd.DataFrame(np.random.randn(nrows, ncols)) for _ in
↳ range(4)]
```

Now let's compare adding them together using plain ol' Python versus `eval()`:

```
In [15]: %timeit df1 + df2 + df3 + df4
11.5 ms +- 845 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

```
In [16]: %timeit pd.eval('df1 + df2 + df3 + df4')
9.79 ms +- 612 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

Now let's do the same thing but with comparisons:

```
In [17]: %timeit (df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)
31.3 ms +- 4.21 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [18]: %timeit pd.eval('(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0)')
10.4 ms +- 469 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

`eval()` also works with unaligned pandas objects:

```
In [19]: s = pd.Series(np.random.randn(50))

In [20]: %timeit df1 + df2 + df3 + df4 + s
21.4 ms +- 2.16 ms per loop (mean +- std. dev. of 7 runs, 10 loops each)
```

```
In [21]: %timeit pd.eval('df1 + df2 + df3 + df4 + s')
9.26 ms +- 788 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

**Note:** Operations such as

```
1 and 2 # would parse to 1 & 2, but should evaluate to 2
3 or 4 # would parse to 3 | 4, but should evaluate to 3
~1 # this is okay, but slower when using eval
```

should be performed in Python. An exception will be raised if you try to perform any boolean/bitwise operations with scalar operands that are not of type `bool` or `np.bool_`. Again, you should perform these kinds of operations in plain Python.

---

### 25.3.3 The `DataFrame.eval` method

In addition to the top level `pandas.eval()` function you can also evaluate an expression in the “context” of a `DataFrame`.

```
In [22]: df = pd.DataFrame(np.random.randn(5, 2), columns=['a', 'b'])

In [23]: df.eval('a + b')
Out[23]:
0    -0.246747
1     0.867786
2    -1.626063
3    -1.134978
4    -1.027798
dtype: float64
```

Any expression that is a valid `pandas.eval()` expression is also a valid `DataFrame.eval()` expression, with the added benefit that you don’t have to prefix the name of the `DataFrame` to the column(s) you’re interested in evaluating.

In addition, you can perform assignment of columns within an expression. This allows for *formulaic evaluation*. The assignment target can be a new column name or an existing column name, and it must be a valid Python identifier.

New in version 0.18.0.

The `inplace` keyword determines whether this assignment will be performed on the original `DataFrame` or return a copy with the new column.

**Warning:** For backwards compatibility, `inplace` defaults to `True` if not specified. This will change in a future version of pandas - if your code depends on an inplace assignment you should update to explicitly set `inplace=True`.

```
In [24]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [25]: df.eval('c = a + b', inplace=True)

In [26]: df.eval('d = a + b + c', inplace=True)

In [27]: df.eval('a = 1', inplace=True)

In [28]: df
Out[28]:
   a  b  c  d
0  1  5  5 10
1  1  6  7 14
```

(continues on next page)



(continued from previous page)

```
2  1  7   9  18
3  1  8  11  22
4  1  9  13  26
```

When `inplace` is set to `False`, a copy of the `DataFrame` with the new or modified columns is returned and the original frame is unchanged.

```
In [29]: df
```

```
Out[29]:
```

```
   a  b  c  d
0  1  5  5 10
1  1  6  7 14
2  1  7  9 18
3  1  8 11 22
4  1  9 13 26
```

```
In [30]: df.eval('e = a - c', inplace=False)
```

```

////////////////////////////////////
↪
   a  b  c  d  e
0  1  5  5 10 -4
1  1  6  7 14 -6
2  1  7  9 18 -8
3  1  8 11 22 -10
4  1  9 13 26 -12
```

```
In [31]: df
```

```

////////////////////////////////////
↪
   a  b  c  d
0  1  5  5 10
1  1  6  7 14
2  1  7  9 18
3  1  8 11 22
4  1  9 13 26
```

New in version 0.18.0.

As a convenience, multiple assignments can be performed by using a multi-line string.

```
In [32]: df.eval("""
.....: c = a + b
.....: d = a + b + c
.....: a = 1""", inplace=False)
.....:
```

```
Out[32]:
```

```
   a  b  c  d
0  1  5  6 12
1  1  6  7 14
2  1  7  8 16
3  1  8  9 18
4  1  9 10 20
```

The equivalent in standard Python would be

```
In [33]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))
```

(continues on next page)

(continued from previous page)

```
In [34]: df['c'] = df.a + df.b
In [35]: df['d'] = df.a + df.b + df.c
In [36]: df['a'] = 1

In [37]: df
Out[37]:
```

	a	b	c	d
0	1	5	5	10
1	1	6	7	14
2	1	7	9	18
3	1	8	11	22
4	1	9	13	26

New in version 0.18.0.

The `query` method gained the `inplace` keyword which determines whether the query modifies the original frame.

```
In [38]: df = pd.DataFrame(dict(a=range(5), b=range(5, 10)))

In [39]: df.query('a > 2')
Out[39]:
```

	a	b
3	3	8
4	4	9

```
In [40]: df.query('a > 2', inplace=True)

In [41]: df
Out[41]:
```

	a	b
3	3	8
4	4	9

**Warning:** Unlike with `eval`, the default value for `inplace` for `query` is `False`. This is consistent with prior versions of pandas.

### 25.3.4 Local Variables

You must *explicitly reference* any local variable that you want to use in an expression by placing the `@` character in front of the name. For example,

```
In [42]: df = pd.DataFrame(np.random.randn(5, 2), columns=list('ab'))

In [43]: newcol = np.random.randn(len(df))

In [44]: df.eval('b + @newcol')
Out[44]:
```

0	-0.173926
1	2.493083
2	-0.881831
3	-0.691045

(continues on next page)

```
In [45]: df.query('b < @newcol')
```



If you don't prefix the local variable with @, pandas will raise an exception telling you the variable is undefined.

```
In [46]: a = np.random.randn()
```

Out [47] :

```
In [48]: df.loc[a < df.a] # same as the previous expression
```

	a	b
0	0.863987	-0.115998

```
In [49]: a, b = 1, 2
```

```
Traceback (most recent call last):
```

```
File "<ipython-input-50-af17947a194f>", line 1, in <module>
    pd.eval('@a + b')
```

```
File "/Users/taugspurger/sandbox/pandas-release/pandas-docs/pandas/core/computation/
→eval.py", line 150, in _check_for_locals
    raise SyntaxError(msg)
```

In this case, you should simply refer to the variables like you would in standard Python.

```
In [51]: pd.eval('a + b')
Out[51]: 3
```

### 25.3.5 pandas.eval() Parsers

There are two different parsers and two different engines you can use as the backend.

The default 'pandas' parser allows a more intuitive syntax for expressing query-like operations (comparisons, conjunctions and disjunctions). In particular, the precedence of the & and | operators is made equal to the precedence of the corresponding boolean operations and and or.

For example, the above conjunction can be written without parentheses. Alternatively, you can use the 'python' parser to enforce strict Python semantics.

```
In [52]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0) '
In [53]: x = pd.eval(expr, parser='python')
In [54]: expr_no_parens = 'df1 > 0 & df2 > 0 & df3 > 0 & df4 > 0'
In [55]: y = pd.eval(expr_no_parens, parser='pandas')
In [56]: np.all(x == y)
Out[56]: True
```

The same expression can be “anded” together with the word `and` as well:

```
In [57]: expr = '(df1 > 0) & (df2 > 0) & (df3 > 0) & (df4 > 0) '
In [58]: x = pd.eval(expr, parser='python')
In [59]: expr_with_ands = 'df1 > 0 and df2 > 0 and df3 > 0 and df4 > 0'
In [60]: y = pd.eval(expr_with_ands, parser='pandas')
In [61]: np.all(x == y)
Out[61]: True
```

The `and` and `or` operators here have the same precedence that they would in vanilla Python.

### 25.3.6 pandas.eval() Backends

There's also the option to make `eval()` operate identical to plain ol' Python.

---

**Note:** Using the 'python' engine is generally *not* useful, except for testing other evaluation engines against it. You will achieve **no** performance benefits using `eval()` with `engine='python'` and in fact may incur a performance hit.

---

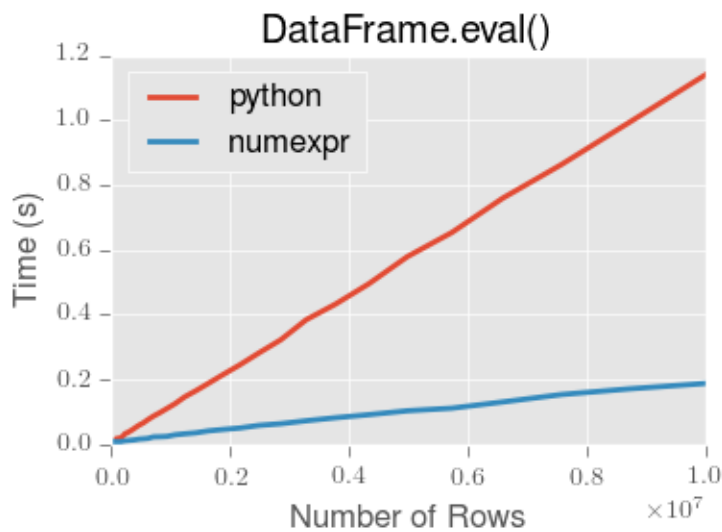
You can see this by using `pandas.eval()` with the 'python' engine. It is a bit slower (not by much) than evaluating the same expression in Python

```
In [62]: %timeit df1 + df2 + df3 + df4
11.7 ms +- 1e+03 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

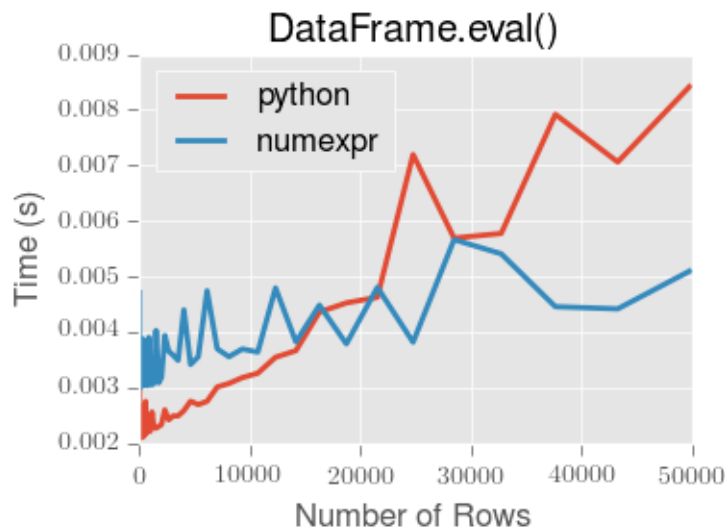
```
In [63]: %timeit pd.eval('df1 + df2 + df3 + df4', engine='python')
12.1 ms +- 811 us per loop (mean +- std. dev. of 7 runs, 100 loops each)
```

### 25.3.7 pandas.eval() Performance

`eval()` is intended to speed up certain kinds of operations. In particular, those operations involving complex expressions with large *DataFrame*/*Series* objects should see a significant performance benefit. Here is a plot showing the running time of `pandas.eval()` as function of the size of the frame involved in the computation. The two lines are two different engines.



**Note:** Operations with smallish objects (around 15k-20k rows) are faster using plain Python:



This plot was created using a *DataFrame* with 3 columns each containing floating point values generated using `numpy.random.randn()`.

### 25.3.8 Technical Minutia Regarding Expression Evaluation

Expressions that would result in an object dtype or involve datetime operations (because of NaT) must be evaluated in Python space. The main reason for this behavior is to maintain backwards compatibility with versions of NumPy < 1.7. In those versions of NumPy a call to `ndarray.astype(str)` will truncate any strings that are more than 60 characters in length. Second, we can't pass object arrays to `numexpr` thus string comparisons must be evaluated in Python space.

The upshot is that this *only* applies to object-dtype'd expressions. So, if you have an expression—for example

```
In [64]: df = pd.DataFrame({'strings': np.repeat(list('cba'), 3),
.....:                    'nums': np.repeat(range(3), 3)})
.....:
```

```
In [65]: df
```

```
Out[65]:
  strings  nums
0        c     0
1        c     0
2        c     0
3        b     1
4        b     1
5        b     1
6        a     2
7        a     2
8        a     2
```

```
In [66]: df.query('strings == "a" and nums == 1')
```

```

////////////////////////////////////
↪
Empty DataFrame
Columns: [strings, nums]
Index: []
```

the numeric part of the comparison (`nums == 1`) will be evaluated by `numexpr`.

In general, `DataFrame.query()`/`pandas.eval()` will evaluate the subexpressions that *can* be evaluated by `numexpr` and those that must be evaluated in Python space transparently to the user. This is done by inferring the result type of an expression from its arguments and operators.

## SPARSE DATA STRUCTURES

---

**Note:** The `SparsePanel` class has been removed in 0.19.0

---

We have implemented “sparse” versions of `Series` and `DataFrame`. These are not sparse in the typical “mostly 0”. Rather, you can view these objects as being “compressed” where any data matching a specific value (`NaN` / missing value, though any value can be chosen) is omitted. A special `SparseIndex` object tracks where data has been “sparsified”. This will make much more sense with an example. All of the standard pandas data structures have a `to_sparse` method:

```
In [1]: ts = pd.Series(randn(10))
In [2]: ts[2:-2] = np.nan
In [3]: sts = ts.to_sparse()
In [4]: sts
Out[4]:
0    0.469112
1   -0.282863
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
7         NaN
8   -0.861849
9   -2.104569
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The `to_sparse` method takes a `kind` argument (for the sparse index, see below) and a `fill_value`. So if we had a mostly zero `Series`, we could convert it to sparse with `fill_value=0`:

```
In [5]: ts.fillna(0).to_sparse(fill_value=0)
Out[5]:
0    0.469112
1   -0.282863
2    0.000000
3    0.000000
4    0.000000
5    0.000000
```

(continues on next page)

(continued from previous page)

```
6      0.000000
7      0.000000
8     -0.861849
9     -2.104569
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA `DataFrame`:

[illegible]

As you can see, the density (% of values that have not been “compressed”) is extremely low. This sparse object takes up much less memory on disk (pickled) and in the Python interpreter. Functionally, their behavior should be nearly identical to their dense counterparts.

Any sparse object can be converted back to the standard dense form by calling `to_dense`:

```
In [11]: sts.to_dense()
Out[11]:
```

0	0.469112
1	-0.282863
2	NaN
3	NaN
4	NaN
5	NaN
6	NaN

(continues on next page)



(continued from previous page)

```

7          NaN
8    -0.861849
9    -2.104569
dtype: float64

```

## 26.1 SparseArray

`SparseArray` is the base layer for all of the sparse indexed data structures. It is a 1-dimensional ndarray-like object storing only values distinct from the `fill_value`:

```

In [12]: arr = np.random.randn(10)

In [13]: arr[2:5] = np.nan; arr[7:8] = np.nan

In [14]: sparr = pd.SparseArray(arr)

In [15]: sparr
Out[15]:
[-1.9556635297215477, -1.6588664275960427, nan, nan, nan, 1.1589328886422277, 0.
↪14529711373305043, nan, 0.6060271905134522, 1.3342113401317768]
Fill: nan
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)

```

Like the indexed objects (`SparseSeries`, `SparseDataFrame`), a `SparseArray` can be converted back to a regular ndarray by calling `to_dense`:

```

In [16]: sparr.to_dense()
Out[16]:
array([-1.9557, -1.6589,      nan,      nan,      nan,  1.1589,  0.1453,
        nan,  0.606 ,  1.3342])

```

## 26.2 SparseIndex objects

Two kinds of `SparseIndex` are implemented, `block` and `integer`. We recommend using `block` as it's more memory efficient. The `integer` format keeps an arrays of all of the locations where the data are not equal to the fill value. The `block` format tracks only the locations and sizes of blocks of data.

## 26.3 Sparse Dtypes

Sparse data should have the same dtype as its dense representation. Currently, `float64`, `int64` and `bool` dtypes are supported. Depending on the original dtype, `fill_value` default changes:

- `float64`: `np.nan`
- `int64`: `0`
- `bool`: `False`

```

In [17]: s = pd.Series([1, np.nan, np.nan])

In [18]: s
Out[18]:
0      1.0
1      NaN
2      NaN
dtype: float64

In [19]: s.to_sparse()
Out[19]:
0      1.0
1      NaN
2      NaN
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [20]: s = pd.Series([1, 0, 0])

In [21]: s
Out[21]:
0      1
1      0
2      0
dtype: int64

In [22]: s.to_sparse()
Out[22]:
0      1
1      0
2      0
dtype: int64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([1], dtype=int32)

In [23]: s = pd.Series([True, False, True])

In [24]: s
Out[24]:
0      True
1     False
2      True
dtype: bool

In [25]: s.to_sparse()
Out[25]:
0      True
1     False
2      True
dtype: bool
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 1], dtype=int32)

```

You can change the dtype using `.astype()`, the result is also sparse. Note that `.astype()` also affects to the



## 26.4 Sparse Calculation

You can apply NumPy *ufuncs* to `SparseArray` and get a `SparseArray` as a result.

```
In [31]: arr = pd.SparseArray([1., np.nan, np.nan, -2., np.nan])

In [32]: np.abs(arr)
Out[32]:
[1.0, nan, nan, 2.0, nan]
Fill: nan
IntIndex
Indices: array([0, 3], dtype=int32)
```

The *ufunc* is also applied to `fill_value`. This is needed to get the correct dense result.

```
In [33]: arr = pd.SparseArray([1., -1, -1, -2., -1], fill_value=-1)

In [34]: np.abs(arr)
Out[34]:
[1.0, 1.0, 1.0, 2.0, 1.0]
Fill: 1
IntIndex
Indices: array([0, 3], dtype=int32)

In [35]: np.abs(arr).to_dense()
Out[35]:
array([ 1.,  1.,  1.,  2.,  1.])
```

## 26.5 Interaction with `scipy.sparse`

### 26.5.1 `SparseDataFrame`

New in version 0.20.0.

Pandas supports creating sparse dataframes directly from `scipy.sparse` matrices.

```
In [36]: from scipy.sparse import csr_matrix

In [37]: arr = np.random.random(size=(1000, 5))

In [38]: arr[arr < .9] = 0

In [39]: sp_arr = csr_matrix(arr)

In [40]: sp_arr
Out[40]:
<1000x5 sparse matrix of type '<class 'numpy.float64''>'
      with 517 stored elements in Compressed Sparse Row format>

In [41]: sdf = pd.SparseDataFrame(sp_arr)

In [42]: sdf
Out[42]:
      0      1      2      3      4
```

(continues on next page)

(continued from previous page)

```

0      0.956380 NaN      NaN      NaN NaN
1           NaN NaN      NaN      NaN NaN
2           NaN NaN      NaN      NaN NaN
3           NaN NaN      NaN      NaN NaN
4      0.999552 NaN      NaN  0.956153 NaN
5           NaN NaN      NaN      NaN NaN
6      0.913638 NaN      NaN      NaN NaN
..      ... ..      ...      ... ..
993      NaN NaN      NaN      NaN NaN
994      NaN NaN      NaN      NaN NaN
995      NaN NaN      NaN  0.998834 NaN
996      NaN NaN      NaN      NaN NaN
997      NaN NaN      NaN      NaN NaN
998      NaN NaN  0.95659      NaN NaN
999      NaN NaN      NaN      NaN NaN

[1000 rows x 5 columns]
```

All sparse formats are supported, but matrices that are not in `COOrdinate` format will be converted, copying data as needed. To convert a `SparseDataFrame` back to sparse SciPy matrix in COO format, you can use the `SparseDataFrame.to_coo()` method:

```

In [43]: sdf.to_coo()
Out[43]:
<1000x5 sparse matrix of type '<class 'numpy.float64'>'
      with 517 stored elements in COOrdinate format>
```

## 26.5.2 SparseSeries

A `SparseSeries.to_coo()` method is implemented for transforming a `SparseSeries` indexed by a `MultiIndex` to a `scipy.sparse.coo_matrix`.

The method requires a `MultiIndex` with two or more levels.

```

In [44]: s = pd.Series([3.0, np.nan, 1.0, 3.0, np.nan, np.nan])

In [45]: s.index = pd.MultiIndex.from_tuples([(1, 2, 'a', 0),
.....:                                     (1, 2, 'a', 1),
.....:                                     (1, 1, 'b', 0),
.....:                                     (1, 1, 'b', 1),
.....:                                     (2, 1, 'b', 0),
.....:                                     (2, 1, 'b', 1)],
.....:                                     names=['A', 'B', 'C', 'D'])

In [46]: s
Out[46]:
A  B  C  D
1  2  a  0    3.0
      1    NaN
      1  b  0    1.0
      1    3.0
2  1  b  0    NaN
      1    NaN
dtype: float64
```

(continues on next page)

(continued from previous page)

```
# SparseSeries
In [47]: ss = s.to_sparse()

In [48]: ss
Out[48]:
A  B  C  D
1  2  a  0    3.0
      1    NaN
      1  b  0    1.0
      1    3.0
2  1  b  0    NaN
      1    NaN

dtype: float64
BlockIndex
Block locations: array([0, 2], dtype=int32)
Block lengths: array([1, 2], dtype=int32)
```

In the example below, we transform the `SparseSeries` to a sparse representation of a 2-d array by specifying that the first and second `MultiIndex` levels define labels for the rows and the third and fourth levels define labels for the columns. We also specify that the column and row labels should be sorted in the final sparse representation.

```
In [49]: A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
.....:                                column_levels=['C', 'D'],
.....:                                sort_labels=True)
.....:
```

```
In [50]: A
Out[50]:
<3x4 sparse matrix of type '<class 'numpy.float64''>'
      with 3 stored elements in COOrdinate format>
```

```
In [51]: A.todense()
//////////
↪
matrix([[ 0.,  0.,  1.,  3.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
```

```
In [53]: columns
\\[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

Specifying different row and column labels (and not sorting them) yields a different sparse matrix:

```
In [54]: A, rows, columns = ss.to_coo(row_levels=['A', 'B', 'C'],
....:                                column_levels=['D'],
....:                                sort_labels=False)
....:
```

```
In [55]: A
Out[55]:
```

(continues on next page)

(continued from previous page)

```
<3x2 sparse matrix of type '<class 'numpy.float64'>'
      with 3 stored elements in COOrdinate format>
```

```
In [56]: A.todense()
```

```
////////////////////////////////////
↪
matrix([[ 3.,  0.],
        [ 1.,  3.],
        [ 0.,  0.]])
```

```
In [57]: rows
```

```
////////////////////////////////////
↪ [(1, 2, 'a'), (1, 1, 'b'), (2, 1, 'b')]
```

```
In [58]: columns
```

```
////////////////////////////////////
↪ [0, 1]
```

A convenience method `SparseSeries.from_coo()` is implemented for creating a `SparseSeries` from a `scipy.sparse.coo_matrix`.

```
In [59]: from scipy import sparse
```

```
In [60]: A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
.....:                        shape=(3, 4))
.....:
```

```
In [61]: A
```

```
Out[61]:
```

```
<3x4 sparse matrix of type '<class 'numpy.float64'>'
      with 3 stored elements in COOrdinate format>
```

```
In [62]: A.todense()
```

```
////////////////////////////////////
↪
matrix([[ 0.,  0.,  1.,  2.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
```

The default behaviour (with `dense_index=False`) simply returns a `SparseSeries` containing only the non-null entries.

```
In [63]: ss = pd.SparseSeries.from_coo(A)
```

```
In [64]: ss
```

```
Out[64]:
```

```
0 2    1.0
  3    2.0
1 0    3.0
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

Specifying `dense_index=True` will result in an index that is the Cartesian product of the row and columns coordinates of the matrix. Note that this will consume a significant amount of memory (relative to `dense_index=False`) if the sparse matrix is large (and sparse) enough.

```
In [65]: ss_dense = pd.SparseSeries.from_coo(A, dense_index=True)
```

```
In [66]: ss_dense
```

```
Out[66]:
```

```
0  0    NaN
   1    NaN
   2    1.0
   3    2.0
1  0    3.0
   1    NaN
   2    NaN
   3    NaN
2  0    NaN
   1    NaN
   2    NaN
   3    NaN
```

```
dtype: float64
```

```
BlockIndex
```

```
Block locations: array([2], dtype=int32)
```

```
Block lengths: array([3], dtype=int32)
```



## FREQUENTLY ASKED QUESTIONS (FAQ)

### 27.1 DataFrame memory usage

The memory usage of a DataFrame (including the index) is shown when calling the `info()`. A configuration option, `display.memory_usage` (see *the list of options*), specifies if the DataFrame's memory usage will be displayed when invoking the `df.info()` method.

For example, the memory usage of the DataFrame below is shown when calling `info()`:

```
In [1]: dtypes = ['int64', 'float64', 'datetime64[ns]', 'timedelta64[ns]',
...:             'complex128', 'object', 'bool']
...:

In [2]: n = 5000

In [3]: data = dict([(t, np.random.randint(100, size=n).astype(t))
...:                 for t in dtypes])
...:

In [4]: df = pd.DataFrame(data)

In [5]: df['categorical'] = df['object'].astype('category')

In [6]: df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
int64          5000 non-null int64
float64        5000 non-null float64
datetime64[ns] 5000 non-null datetime64[ns]
timedelta64[ns] 5000 non-null timedelta64[ns]
complex128     5000 non-null complex128
object         5000 non-null object
bool           5000 non-null bool
categorical    5000 non-null category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↪object(1), timedelta64[ns](1)
memory usage: 289.1+ KB
```

The + symbol indicates that the true memory usage could be higher, because pandas does not count the memory used by values in columns with `dtype=object`.

Passing `memory_usage='deep'` will enable a more accurate memory usage report, accounting for the full usage of the contained objects. This is optional as it can be expensive to do this deeper introspection.

```
In [7]: df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 8 columns):
int64          5000 non-null int64
float64        5000 non-null float64
datetime64[ns] 5000 non-null datetime64[ns]
timedelta64[ns] 5000 non-null timedelta64[ns]
complex128      5000 non-null complex128
object         5000 non-null object
bool           5000 non-null bool
category       5000 non-null category
dtypes: bool(1), category(1), complex128(1), datetime64[ns](1), float64(1), int64(1),
↳ object(1), timedelta64[ns](1)
memory usage: 425.6 KB
```

By default the display option is set to `True` but can be explicitly overridden by passing the `memory_usage` argument when invoking `df.info()`.

The memory usage of each column can be found by calling the `memory_usage()` method. This returns a Series with an index represented by column names and memory usage of each column shown in bytes. For the DataFrame above, the memory usage of each column and the total memory usage can be found with the `memory_usage` method:

```
In [8]: df.memory_usage()
```

Out [8] :

Index	80
int64	40000
float64	40000
datetime64[ns]	40000
timedelta64[ns]	40000
complex128	80000
object	40000
bool	5000
categorical	10920
dtype: int64	

```
# total memory usage of dataframe
```

```
In [9]: df.memory_usage().sum()
```

$\backslash \backslash \backslash \backslash \backslash$   
→ 296000

By default the memory usage of the `DataFrame`'s index is shown in the returned `Series`, the memory usage of the index can be suppressed by passing the `index=False` argument:

```
In [10]: df.memory_usage(index=False)
```

Out [10] :

int64	40000
float64	40000
datetime64[ns]	40000
timedelta64[ns]	40000
complex128	80000
object	40000
bool	5000
categorical	10920
dtype: int64	

The memory usage displayed by the `info()` method utilizes the `memory_usage()` method to determine the memory usage of a `DataFrame` while also formatting the output in human-readable units (base-2 representation; i.e. 1KB

= 1024 bytes).

See also *Categorical Memory Usage*.

## 27.2 Using If/Truth Statements with pandas

pandas follows the NumPy convention of raising an error when you try to convert something to a `bool`. This happens in an `if`-statement or when using the boolean operations: `and`, `or`, and `not`. It is not clear what the result of the following code should be:

```
>>> if pd.Series([False, True, False]):
    ...
```

Should it be `True` because it's not zero-length, or `False` because there are `False` values? It is unclear, so instead, pandas raises a `ValueError`:

```
>>> if pd.Series([False, True, False]):
    print("I was true")
Traceback
...
ValueError: The truth value of an array is ambiguous. Use a.empty, a.any() or a.all().
```

You need to explicitly choose what you want to do with the `DataFrame`, e.g. use `any()`, `all()` or `empty()`. Alternatively, you might want to compare if the pandas object is `None`:

```
>>> if pd.Series([False, True, False]) is not None:
    print("I was not None")
>>> I was not None
```

Below is how to check if any of the values are `True`:

```
>>> if pd.Series([False, True, False]).any():
    print("I am any")
>>> I am any
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```
In [11]: pd.Series([True]).bool()
Out[11]: True

In [12]: pd.Series([False]).bool()
Out[12]: False

In [13]: pd.DataFrame([True]).bool()
Out[13]: True

In [14]: pd.DataFrame([False]).bool()
Out[14]: False
```

### 27.2.1 Bitwise boolean

Bitwise boolean operators like `==` and `!=` return a boolean `Series`, which is almost always what you want anyways.

```
>>> s = pd.Series(range(5))
>>> s == 4
0    False
1    False
2    False
3    False
4     True
dtype: bool
```

See *boolean comparisons* for more examples.

### 27.2.2 Using the `in` operator

Using the Python `in` operator on a `Series` tests for membership in the index, not membership among the values.

```
In [15]: s = pd.Series(range(5), index=list('abcde'))

In [16]: 2 in s
Out[16]: False

In [17]: 'b' in s
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Out[17]: True
```

If this behavior is surprising, keep in mind that using `in` on a Python dictionary tests keys, not values, and `Series` are dict-like. To test for membership in the values, use the method `isin()`:

```
In [18]: s.isin([2])
Out[18]:
a    False
b    False
c     True
d    False
e    False
dtype: bool

In [19]: s.isin([2]).any()
Out[19]: True
```

For DataFrames, likewise, `in` applies to the column axis, testing for membership in the list of column names.

### 27.3 NaN, Integer NA values and NA type promotions

### 27.3.1 Choice of NA representation

For lack of NA (missing) support from the ground up in NumPy and Python in general, we were given the difficult choice between either:

- A *masked array* solution: an array of data and an array of boolean values indicating whether a value is there or is missing.
- Using a special sentinel value, bit pattern, or set of sentinel values to denote NA across the dtypes.

For many reasons we chose the latter. After years of production use it has proven, at least in my opinion, to be the best decision given the state of affairs in NumPy and Python in general. The special value `NaN` (Not-A-Number) is used

everywhere as the NA value, and there are API functions `isna` and `notna` which can be used across the dtypes to detect NA values.

However, it comes with it a couple of trade-offs which I most certainly have not ignored.

### 27.3.2 Support for integer NA

In the absence of high performance NA support being built into NumPy from the ground up, the primary casualty is the ability to represent NAs in integer arrays. For example:

```
In [20]: s = pd.Series([1, 2, 3, 4, 5], index=list('abcde'))

In [21]: s
Out[21]:
a      1
b      2
c      3
d      4
e      5
dtype: int64

In [22]: s.dtype
Out[22]: dtype('int64')

In [23]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])

In [24]: s2
Out[24]:
a      1.0
b      2.0
c      3.0
f      NaN
u      NaN
dtype: float64

In [25]: s2.dtype
Out[25]: dtype('float64')
```

This trade-off is made largely for memory and performance reasons, and also so that the resulting `Series` continues to be “numeric”. One possibility is to use `dtype=object` arrays instead.

### 27.3.3 NA type promotions

When introducing NAs into an existing `Series` or `DataFrame` via `reindex()` or some other means, boolean and integer types will be promoted to a different dtype in order to store the NAs. The promotions are summarized in this table:

Typeclass	Promotion dtype for storing NAs
floating	no change
object	no change
integer	cast to float64
boolean	cast to object

While this may seem like a heavy trade-off, I have found very few cases where this is an issue in practice i.e. storing values greater than  $2^{53}$ . Some explanation for the motivation is in the next section.

### 27.3.4 Why not make NumPy like R?

Many people have suggested that NumPy should simply emulate the NA support present in the more domain-specific statistical programming language R. Part of the reason is the NumPy type hierarchy:

Typeclass	Dtypes
<code>numpy.floating</code>	<code>float16</code> , <code>float32</code> , <code>float64</code> , <code>float128</code>
<code>numpy.integer</code>	<code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code>
<code>numpy.unsignedinteger</code>	<code>uint8</code> , <code>uint16</code> , <code>uint32</code> , <code>uint64</code>
<code>numpy.object_</code>	<code>object_</code>
<code>numpy.bool_</code>	<code>bool_</code>
<code>numpy.character</code>	<code>string_</code> , <code>unicode_</code>

The R language, by contrast, only has a handful of built-in data types: `integer`, `numeric` (floating-point), `character`, and `boolean`. NA types are implemented by reserving special bit patterns for each type to be used as the missing value. While doing this with the full NumPy type hierarchy would be possible, it would be a more substantial trade-off (especially for the 8- and 16-bit data types) and implementation undertaking.

An alternate approach is that of using masked arrays. A masked array is an array of data with an associated boolean *mask* denoting whether each value should be considered NA or not. I am personally not in love with this approach as I feel that overall it places a fairly heavy burden on the user and the library implementer. Additionally, it exacts a fairly high performance cost when working with numerical data compared with the simple approach of using NaN. Thus, I have chosen the Pythonic “practicality beats purity” approach and traded integer NA capability for a much simpler approach of using a special value in float and object arrays to denote NA, and promoting integer arrays to floating when NAs must be introduced.

## 27.4 Differences with NumPy

For `Series` and `DataFrame` objects, `var()` normalizes by  $N-1$  to produce unbiased estimates of the sample variance, while NumPy’s `var` normalizes by  $N$ , which measures the variance of the sample. Note that `cov()` normalizes by  $N-1$  in both pandas and NumPy.

## 27.5 Thread-safety

As of pandas 0.11, pandas is not 100% thread safe. The known issues relate to the `copy()` method. If you are doing a lot of copying of `DataFrame` objects shared among threads, we recommend holding locks inside the threads where the data copying occurs.

See [this link](#) for more information.

## 27.6 Byte-Ordering Issues

Occasionally you may have to deal with data that were created on a machine with a different byte order than the one on which you are running Python. A common symptom of this issue is an error like:

```
Traceback
...
ValueError: Big-endian buffer not supported on little-endian compiler
```

To deal with this issue you should convert the underlying NumPy array to the native system byte order *before* passing it to Series or DataFrame constructors using something similar to the following:

```
In [26]: x = np.array(list(range(10)), '>i4') # big endian

In [27]: newx = x.byteswap().newbyteorder() # force native byteorder

In [28]: s = pd.Series(newx)
```

See [the NumPy documentation on byte order](#) for more details.





## RPY2 / R INTERFACE

**Warning:** Up to pandas 0.19, a `pandas.rpy` module existed with functionality to convert between pandas and rpy2 objects. This functionality now lives in the rpy2 project itself. See the [updating section](#) of the previous documentation for a guide to port your code from the removed `pandas.rpy` to rpy2 functions.

rpy2 is an interface to R running embedded in a Python process, and also includes functionality to deal with pandas DataFrames. Converting data frames back and forth between rpy2 and pandas should be largely automated (no need to convert explicitly, it will be done on the fly in most rpy2 functions). To convert explicitly, the functions are `pandas2ri.py2ri()` and `pandas2ri.ri2py()`.

See also the documentation of the rpy2 project: <https://rpy2.readthedocs.io>.

In the remainder of this page, a few examples of explicit conversion is given. The pandas conversion of rpy2 needs first to be activated:

```
In [1]: from rpy2.robjjects import r, pandas2ri
In [2]: pandas2ri.activate()
```

### 28.1 Transferring R data sets into Python

Once the pandas conversion is activated (`pandas2ri.activate()`), many conversions of R to pandas objects will be done automatically. For example, to obtain the ‘iris’ dataset as a pandas DataFrame:

```
In [3]: r.data('iris')
Out[3]:
R object with classes: ('character',) mapped to:
<StrVector - Python:0x1c3987a8c8 / R:0x7fb4bc93d8e8>
['iris']
```

```
In [4]: r['iris'].head()
```

```
=====
↪
   Sepal.Length  Sepal.Width  Petal.Length  Petal.Width Species
0             5.1           3.5           1.4           0.2  setosa
1             4.9           3.0           1.4           0.2  setosa
2             4.7           3.2           1.3           0.2  setosa
3             4.6           3.1           1.5           0.2  setosa
4             5.0           3.6           1.4           0.2  setosa
```

If the pandas conversion was not activated, the above could also be accomplished by explicitly converting it with the `pandas2ri.ri2py` function (`pandas2ri.ri2py(r['iris'])`).

## 28.2 Converting DataFrames into R objects

The `pandas2ri.py2ri` function support the reverse operation to convert DataFrames into the equivalent R object (that is, **data.frame**):

```
In [5]: df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]},
...:                      index=["one", "two", "three"])
...:

In [6]: r_dataframe = pandas2ri.py2ri(df)

In [7]: print(type(r_dataframe))
<class 'rpy2.robjects.vectors.DataFrame'>

In [8]: print(r_dataframe)
\\repeated backslashes\\           A B C
one    1 4 7
two     2 5 8
three   3 6 9
```

The DataFrame's index is stored as the `rownames` attribute of the `data.frame` instance.

## PANDAS ECOSYSTEM

Increasingly, packages are being built on top of pandas to address specific needs in data preparation, analysis and visualization. This is encouraging because it means pandas is not only helping users to handle their data tasks but also that it provides a better starting point for developers to build powerful and more focused data tools. The creation of libraries that complement pandas' functionality also allows pandas development to remain focused around its original requirements.

This is an in-exhaustive list of projects that build on pandas in order to provide tools in the PyData space.

We'd like to make it easier for users to find these project, if you know of other substantial projects that you feel should be on this list, please let us know.

### 29.1 Statistics and Machine Learning

#### 29.1.1 Statsmodels

Statsmodels is the prominent Python "statistics and econometrics library" and it has a long-standing special relationship with pandas. Statsmodels provides powerful statistics, econometrics, analysis and modeling functionality that is out of pandas' scope. Statsmodels leverages pandas objects as the underlying data container for computation.

#### 29.1.2 sklearn-pandas

Use pandas DataFrames in your [scikit-learn](#) ML pipeline.

#### 29.1.3 Featuretools

Featuretools is a Python library for automated feature engineering built on top of pandas. It excels at transforming temporal and relational datasets into feature matrices for machine learning using reusable feature engineering "primitives". Users can contribute their own primitives in Python and share them with the rest of the community.

### 29.2 Visualization

#### 29.2.1 Bokeh

Bokeh is a Python interactive visualization library for large datasets that natively uses the latest web technologies. Its goal is to provide elegant, concise construction of novel graphics in the style of Protovis/D3, while delivering high-performance interactivity over large data to thin clients.

## 29.2.2 seaborn

Seaborn is a Python visualization library based on [matplotlib](#). It provides a high-level, dataset-oriented interface for creating attractive statistical graphics. The plotting functions in seaborn understand pandas objects and leverage pandas grouping operations internally to support concise specification of complex visualizations. Seaborn also goes beyond matplotlib and pandas with the option to perform statistical estimation while plotting, aggregating across observations and visualizing the fit of statistical models to emphasize patterns in a dataset.

## 29.2.3 yhat/ggplot

Hadley Wickham's [ggplot2](#) is a foundational exploratory visualization package for the R language. Based on “[The Grammar of Graphics](#)” it provides a powerful, declarative and extremely general way to generate bespoke plots of any kind of data. It's really quite incredible. Various implementations to other languages are available, but a faithful implementation for Python users has long been missing. Although still young (as of Jan-2014), the [yhat/ggplot](#) project has been progressing quickly in that direction.

## 29.2.4 Vincent

The [Vincent](#) project leverages [Vega](#) (that in turn, leverages [d3](#)) to create plots. Although functional, as of Summer 2016 the Vincent project has not been updated in over two years and is [unlikely to receive further updates](#).

## 29.2.5 IPython Vega

Like Vincent, the [IPython Vega](#) project leverages [Vega](#) to create plots, but primarily targets the IPython Notebook environment.

## 29.2.6 Plotly

[Plotly's Python API](#) enables interactive figures and web shareability. Maps, 2D, 3D, and live-streaming graphs are rendered with WebGL and [D3.js](#). The library supports plotting directly from a pandas DataFrame and cloud-based collaboration. Users of [matplotlib](#), [ggplot for Python](#), and [Seaborn](#) can convert figures into interactive web-based plots. Plots can be drawn in [IPython Notebooks](#), edited with R or MATLAB, modified in a GUI, or embedded in apps and dashboards. Plotly is free for unlimited sharing, and has [cloud](#), [offline](#), or [on-premise](#) accounts for private use.

## 29.2.7 QtPandas

Spun off from the main pandas library, the [qtpandas](#) library enables DataFrame visualization and manipulation in PyQt4 and PySide applications.

# 29.3 IDE

## 29.3.1 IPython

IPython is an interactive command shell and distributed computing environment. IPython Notebook is a web application for creating IPython notebooks. An IPython notebook is a JSON document containing an ordered list of input/output cells which can contain code, text, mathematics, plots and rich media. IPython notebooks can be converted to a number of open standard output formats (HTML, HTML presentation slides, LaTeX, PDF, ReStructuredText, Markdown, Python) through ‘Download As’ in the web interface and `ipython nbconvert` in a shell.

Pandas DataFrames implement `_repr_html_` methods which are utilized by IPython Notebook for displaying (abbreviated) HTML tables. (Note: HTML tables may or may not be compatible with non-HTML IPython output formats.)

### 29.3.2 quantopian/qgrid

qgrid is “an interactive grid for sorting and filtering DataFrames in IPython Notebook” built with SlickGrid.

### 29.3.3 Spyder

Spyder is a cross-platform Qt-based open-source Python IDE with editing, testing, debugging, and introspection features. Spyder can now introspect and display Pandas DataFrames and show both “column wise min/max and global min/max coloring.”

## 29.4 API

### 29.4.1 pandas-datareader

`pandas-datareader` is a remote data access library for pandas (PyPI:`pandas-datareader`). It is based on functionality that was located in `pandas.io.data` and `pandas.io.wb` but was split off in v0.19. See more in the [pandas-datareader docs](#):

The following data feeds are available:

- Yahoo! Finance
- Google Finance
- FRED
- Fama/French
- World Bank
- OECD
- Eurostat
- EDGAR Index

### 29.4.2 quandl/Python

Quandl API for Python wraps the Quandl REST API to return Pandas DataFrames with timeseries indexes.

### 29.4.3 pydatastream

PyDatastream is a Python interface to the [Thomson Dataworks Enterprise \(DWE/Datastream\)](#) SOAP API to return indexed Pandas DataFrames or Panels with financial data. This package requires valid credentials for this API (non free).

## 29.4.4 pandaSDMX

pandaSDMX is a library to retrieve and acquire statistical data and metadata disseminated in [SDMX 2.1](#), an ISO-standard widely used by institutions such as statistics offices, central banks, and international organisations. pandaSDMX can expose datasets and related structural metadata including dataflows, code-lists, and datastructure definitions as pandas Series or multi-indexed DataFrames.

## 29.4.5 fredapi

fredapi is a Python interface to the [Federal Reserve Economic Data \(FRED\)](#) provided by the Federal Reserve Bank of St. Louis. It works with both the FRED database and ALFRED database that contains point-in-time data (i.e. historic data revisions). fredapi provides a wrapper in Python to the FRED HTTP API, and also provides several convenient methods for parsing and analyzing point-in-time data from ALFRED. fredapi makes use of pandas and returns data in a Series or DataFrame. This module requires a FRED API key that you can obtain for free on the FRED website.

# 29.5 Domain Specific

## 29.5.1 Geopandas

Geopandas extends pandas data objects to include geographic information which support geometric operations. If your work entails maps and geographical coordinates, and you love pandas, you should take a close look at Geopandas.

## 29.5.2 xarray

xarray brings the labeled data power of pandas to the physical sciences by providing N-dimensional variants of the core pandas data structures. It aims to provide a pandas-like and pandas-compatible toolkit for analytics on multi-dimensional arrays, rather than the tabular data for which pandas excels.

# 29.6 Out-of-core

## 29.6.1 Dask

Dask is a flexible parallel computing library for analytics. Dask provides a familiar `DataFrame` interface for out-of-core, parallel and distributed computing.

## 29.6.2 Dask-ML

Dask-ML enables parallel and distributed machine learning using Dask alongside existing machine learning libraries like Scikit-Learn, XGBoost, and TensorFlow.

## 29.6.3 Blaze

Blaze provides a standard API for doing computations with various in-memory and on-disk backends: NumPy, Pandas, SQLAlchemy, MongoDB, PyTables, PySpark.

### 29.6.4 Odo

Odo provides a uniform API for moving data between different formats. It uses pandas own `read_csv` for CSV IO and leverages many existing packages such as PyTables, h5py, and pymongo to move data between non pandas formats. Its graph based approach is also extensible by end users for custom formats that may be too specific for the core of odo.

## 29.7 Data validation

### 29.7.1 Engarde

Engarde is a lightweight library used to explicitly state your assumptions about your datasets and check that they're *actually* true.

## 29.8 Extension Data Types

Pandas provides an interface for defining *extension types* to extend NumPy's type system. The following libraries implement that interface to provide types not found in NumPy or pandas, which work well with pandas' data containers.

### 29.8.1 cyberpandas

Cyberpandas provides an extension type for storing arrays of IP Addresses. These arrays can be stored inside pandas' Series and DataFrame.

## 29.9 Accessors

A directory of projects providing *extension accessors*. This is for users to discover new accessors and for library authors to coordinate on the namespace.

Library	Accessor	Classes
<a href="#">cyberpandas</a>	ip	Series
<a href="#">pdvega</a>	vgplot	Series, DataFrame





## COMPARISON WITH R / R LIBRARIES

Since `pandas` aims to provide a lot of the data manipulation and analysis functionality that people use `R` for, this page was started to provide a more detailed look at the `R` language and its many third party libraries as they relate to `pandas`. In comparisons with `R` and CRAN libraries, we care about the following things:

- **Functionality / flexibility:** what can/cannot be done with each tool
- **Performance:** how fast are operations. Hard numbers/benchmarks are preferable
- **Ease-of-use:** Is one tool easier/harder to use (you may have to be the judge of this, given side-by-side code comparisons)

This page is also here to offer a bit of a translation guide for users of these `R` packages.

For transfer of `DataFrame` objects from `pandas` to `R`, one option is to use `HDF5` files, see [External Compatibility](#) for an example.

### 30.1 Quick Reference

We'll start off with a quick reference guide pairing some common `R` operations using `dplyr` with `pandas` equivalents.

#### 30.1.1 Querying, Filtering, Sampling

R	pandas
<code>dim(df)</code>	<code>df.shape</code>
<code>head(df)</code>	<code>df.head()</code>
<code>slice(df, 1:10)</code>	<code>df.iloc[:9]</code>
<code>filter(df, col1 == 1, col2 == 1)</code>	<code>df.query('col1 == 1 &amp; col2 == 1')</code>
<code>df[df\$col1 == 1 &amp; df\$col2 == 1,]</code>	<code>df[(df.col1 == 1) &amp; (df.col2 == 1)]</code>
<code>select(df, col1, col2)</code>	<code>df[['col1', 'col2']]</code>
<code>select(df, col1:col3)</code>	<code>df.loc[:, 'col1':'col3']</code>
<code>select(df, -(col1:col3))</code>	<code>df.drop(cols_to_drop, axis=1)</code> but see <sup>1</sup>
<code>distinct(select(df, col1))</code>	<code>df[['col1']].drop_duplicates()</code>
<code>distinct(select(df, col1, col2))</code>	<code>df[['col1', 'col2']].drop_duplicates()</code>
<code>sample_n(df, 10)</code>	<code>df.sample(n=10)</code>
<code>sample_frac(df, 0.01)</code>	<code>df.sample(frac=0.01)</code>

---

<sup>1</sup> `R`'s shorthand for a subrange of columns (`select(df, col1:col3)`) can be approached cleanly in `pandas`, if you have the list of columns, for example `df[cols[1:3]]` or `df.drop(cols[1:3])`, but doing this by column name is a bit messy.

### 30.1.2 Sorting

R	pandas
<code>arrange(df, col1, col2)</code>	<code>df.sort_values(['col1', 'col2'])</code>
<code>arrange(df, desc(col1))</code>	<code>df.sort_values('col1', ascending=False)</code>

### 30.1.3 Transforming

R	pandas
<code>select(df, col_one = col1)</code>	<code>df.rename(columns={'col1': 'col_one'})['col_one']</code>
<code>rename(df, col_one = col1)</code>	<code>df.rename(columns={'col1': 'col_one'})</code>
<code>mutate(df, c=a-b)</code>	<code>df.assign(c=df.a-df.b)</code>

### 30.1.4 Grouping and Summarizing

R	pandas
<code>summary(df)</code>	<code>df.describe()</code>
<code>gdf &lt;- group_by(df, col1)</code>	<code>gdf = df.groupby('col1')</code>
<code>summarise(gdf, avg=mean(col1, na.rm=TRUE))</code>	<code>df.groupby('col1').agg({'col1': 'mean'})</code>
<code>summarise(gdf, total=sum(col1))</code>	<code>df.groupby('col1').sum()</code>

## 30.2 Base R

### 30.2.1 Slicing with R's `c`

R makes it easy to access `data.frame` columns by name

```
df <- data.frame(a=rnorm(5), b=rnorm(5), c=rnorm(5), d=rnorm(5), e=rnorm(5))
df[, c("a", "c", "e")]
```

or by integer location

```
df <- data.frame(matrix(rnorm(1000), ncol=100))
df[, c(1:10, 25:30, 40, 50:100)]
```

Selecting multiple columns by name in pandas is straightforward

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list('abc'))

In [2]: df[['a', 'c']]
Out[2]:
```

	a	c
0	-1.039575	-0.424972
1	0.567020	-1.087401
2	-0.673690	-1.478427

(continues on next page)

(continued from previous page)

```

3  0.524988  0.577046
4 -1.715002 -0.370647
5 -1.157892  0.844885
6  1.075770  1.643563
7 -1.469388 -0.674600
8 -1.776904 -1.294524
9  0.413738 -0.472035

```

```
In [3]: df.loc[:, ['a', 'c']]
```

```

↪
      a      c
0 -1.039575 -0.424972
1  0.567020 -1.087401
2 -0.673690 -1.478427
3  0.524988  0.577046
4 -1.715002 -0.370647
5 -1.157892  0.844885
6  1.075770  1.643563
7 -1.469388 -0.674600
8 -1.776904 -1.294524
9  0.413738 -0.472035

```

Selecting multiple noncontiguous columns by integer location can be achieved with a combination of the `iloc` indexer attribute and `numpy.r_`.

```
In [4]: named = list('abcdefg')
```

```
In [5]: n = 30
```

```
In [6]: columns = named + np.arange(len(named), n).tolist()
```

```
In [7]: df = pd.DataFrame(np.random.randn(n, n), columns=columns)
```

```
In [8]: df.iloc[:, np.r_[10, 24:30]]
```

```

Out[8]:
      a      b      c      d      e      ...      25      26
↪      27      28      29
0 -0.013960 -0.362543 -0.006154 -0.923061  0.895717  ... -2.211372  0.974466 -2.
↪ 006747 -0.410001 -0.078638
1  0.545952 -1.219217 -1.226825  0.769804 -1.281247  ... -0.826591 -0.345352  1.
↪ 314232  0.690579  0.995761
2  2.396780  0.014871  3.357427 -0.317441 -1.236269  ...  0.299368 -0.863838  0.
↪ 408204 -1.048089 -0.025747
3 -0.988387  0.094055  1.262731  1.289997  0.082423  ... -0.744471  0.758527  1.
↪ 729689 -0.964980 -0.845696
4 -1.340896  1.846883 -1.328865  1.682706 -1.717693  ... -1.461665 -1.137707 -0.
↪ 891060 -0.693921  1.613616
5  0.464000  0.227371 -0.496922  0.306389 -2.290613  ...  1.952541 -1.056652  0.
↪ 533946 -1.226970  0.040403
6 -0.507516 -0.230096  0.394500 -1.934370 -1.652499  ... -1.833722  1.771740 -0.
↪ 670027  0.049307 -0.521493
..      ...      ...      ...      ...      ...      ...      ...
↪      ...      ...      ...
23 -0.083272 -0.273955 -0.772369 -1.242807 -0.386336  ... -0.873585 -0.699862  0.
↪ 812477 -0.469503  1.142702

```

(continues on next page)

(continued from previous page)

```

24  2.071413 -1.364763  1.122066  0.066847  1.751987  ...  -0.796211  0.241596  0.
↪385922 -0.486078  0.433042
25  0.036609  0.359986  1.211905  0.850427  1.554957  ...  -2.513465 -0.139184  0.
↪810491  0.571599 -0.000676
26 -1.179240  0.238923  1.756671 -0.747571  0.543625  ...   0.307941  1.809049  0.
↪296237 -0.143550  0.289401
27  0.025645  0.932436 -1.694531 -0.182236 -1.072710  ...  -3.060395  0.040268  0.
↪066091 -0.192862  1.979055
28  0.439086  0.812684 -0.128932 -0.142506 -1.137207  ...   0.869610  0.364726 -0.
↪226101 -0.657647 -0.952699
29 -0.909806 -0.312006  0.383630 -0.631606  1.321415  ...   0.211283  1.440190 -0.
↪989193  0.313335 -0.399709

[30 rows x 16 columns]

```

### 30.2.2 aggregate

In R you may want to split data into subsets and compute the mean for each. Using a data.frame called `df` and splitting it into groups `by1` and `by2`:

```

df <- data.frame(
  v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
  v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99),
  by1 = c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12),
  by2 = c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA))
aggregate(x=df[, c("v1", "v2")], by=list(mydf2$by1, mydf2$by2), FUN = mean)

```

The `groupby()` method is similar to base R `aggregate` function.

```

In [9]: df = pd.DataFrame({
...:     'v1': [1,3,5,7,8,3,5,np.nan,4,5,7,9],
...:     'v2': [11,33,55,77,88,33,55,np.nan,44,55,77,99],
...:     'by1': ["red", "blue", 1, 2, np.nan, "big", 1, 2, "red", 1, np.nan, 12],
...:     'by2': ["wet", "dry", 99, 95, np.nan, "damp", 95, 99, "red", 99, np.nan,
...:             np.nan]
...: })
...:

In [10]: g = df.groupby(['by1', 'by2'])

In [11]: g[['v1', 'v2']].mean()
Out[11]:

```

		v1	v2
by1	by2		
1	95	5.0	55.0
	99	5.0	55.0
2	95	7.0	77.0
	99	NaN	NaN
big	damp	3.0	33.0
blue	dry	3.0	33.0
red	red	4.0	44.0
	wet	1.0	11.0

For more details and examples see [the groupby documentation](#).

### 30.2.3 match / %in%

A common way to select data in R is using `%in%` which is defined using the function `match`. The operator `%in%` is used to return a logical vector indicating if there is a match or not:

```
s <- 0:4
s %in% c(2,4)
```

The `isin()` method is similar to R `%in%` operator:

```
In [12]: s = pd.Series(np.arange(5), dtype=np.float32)

In [13]: s.isin([2, 4])
Out[13]:
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

The `match` function returns a vector of the positions of matches of its first argument in its second:

```
s <- 0:4
match(s, c(2,4))
```

For more details and examples see [the reshaping documentation](#).

### 30.2.4 tapply

`tapply` is similar to `aggregate`, but data can be in a ragged array, since the subclass sizes are possibly irregular. Using a data.frame called `baseball`, and retrieving information based on the array `team`:

```
baseball <-
  data.frame(team = gl(5, 5,
    labels = paste("Team", LETTERS[1:5])),
    player = sample(letters, 25),
    batting.average = runif(25, .200, .400))

tapply(baseball$batting.average, baseball$team,
  max)
```

In pandas we may use `pivot_table()` method to handle this:

```
In [14]: import random

In [15]: import string

In [16]: baseball = pd.DataFrame({
....:     'team': ["team %d" % (x+1) for x in range(5)]*5,
....:     'player': random.sample(list(string.ascii_lowercase), 25),
....:     'batting avg': np.random.uniform(.200, .400, 25)
....: })

In [17]: baseball.pivot_table(values='batting avg', columns='team', aggfunc=np.max)
```

(continues on next page)

(continued from previous page)

```
Out[17]:
```

team	team 1	team 2	team 3	team 4	team 5
batting avg	0.394457	0.39573	0.343015	0.388863	0.377379

For more details and examples see [the reshaping documentation](#).

### 30.2.5 subset

The `query()` method is similar to the base R `subset` function. In R you might want to get the rows of a data frame where one column's values are less than another column's values:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,] # note the comma
```

In pandas, there are a few ways to perform subsetting. You can use `query()` or pass an expression as if it were an index/slice as well as standard boolean indexing:

```
In [18]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})
```

```
In [19]: df.query('a <= b')
```

```
Out [19]:
```

	a	b
0	-1.003455	-0.990738
1	0.083515	0.548796
3	-0.524392	0.904400
4	-0.837804	0.746374
8	-0.507219	0.245479

```
In [20]: df[df.a <= df.b]
```

	a	b
0	-1.003455	-0.990738
1	0.083515	0.548796
3	-0.524392	0.904400
4	-0.837804	0.746374
8	-0.507219	0.245479

```
In [21]: df.loc[df.a <= df.b]
```

	a	b
0	-1.003455	-0.990738
1	0.083515	0.548796
3	-0.524392	0.904400
4	-0.837804	0.746374
8	-0.507219	0.245479

For more details and examples see [the query documentation](#).

### 30.2.6 with

An expression using a data.frame called `df` in R with the columns `a` and `b` would be evaluated using `with` like so:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
with(df, a + b)
df$a + df$b # same as the previous expression
```

In pandas the equivalent expression, using the `eval()` method, would be:

```
In [22]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [23]: df.eval('a + b')
Out[23]:
0    -0.920205
1    -0.860236
2     1.154370
3     0.188140
4    -1.163718
5     0.001397
6    -0.825694
7    -1.138198
8    -1.708034
9     1.148616
dtype: float64
```

```
In [24]: df.a + df.b # same as the previous expression
```

```
0 -0.920205
1 -0.860236
2  1.154370
3  0.188140
4 -1.163718
5  0.001397
6 -0.825694
7 -1.138198
8 -1.708034
9  1.148616
dtype: float64
```

In certain cases `eval()` will be much faster than evaluation in pure Python. For more details and examples see [the eval documentation](#).

### 30.3 plyr

`plyr` is an R library for the split-apply-combine strategy for data analysis. The functions revolve around three data structures in R, `a` for arrays, `l` for lists, and `d` for `data.frame`. The table below shows how these data structures could be mapped in Python.

R	Python
array	list
lists	dictionary or list of objects
data.frame	dataframe

### 30.3.1 ddply

An expression using a data.frame called `df` in R where you want to summarize `x` by month:

```
require(plyr)
df <- data.frame(
  x = runif(120, 1, 168),
  y = runif(120, 7, 334),
  z = runif(120, 1.7, 20.7),
  month = rep(c(5,6,7,8),30),
  week = sample(1:4, 120, TRUE)
)

ddply(df, .(month, week), summarize,
      mean = round(mean(x), 2),
      sd = round(sd(x), 2))
```

In pandas the equivalent expression, using the `groupby()` method, would be:

```
In [25]: df = pd.DataFrame({
.....:     'x': np.random.uniform(1., 168., 120),
.....:     'y': np.random.uniform(7., 334., 120),
.....:     'z': np.random.uniform(1.7, 20.7, 120),
.....:     'month': [5,6,7,8]*30,
.....:     'week': np.random.randint(1,4, 120)
.....: })

In [26]: grouped = df.groupby(['month','week'])

In [27]: grouped['x'].agg([np.mean, np.std])
Out[27]:
```

month	week	mean	std
5	1	71.840596	52.886392
	2	71.904794	55.786805
	3	89.845632	49.892367
6	1	97.730877	52.442172
	2	93.369836	47.178389
	3	96.592088	58.773744
7	1	59.255715	43.442336
	2	69.634012	28.607369
	3	84.510992	59.761096
8	1	104.787666	31.745437
	2	69.717872	53.747188
	3	79.892221	52.950459

For more details and examples see [the groupby documentation](#).

## 30.4 reshape / reshape2

### 30.4.1 melt.array

An expression using a 3 dimensional array called `a` in R where you want to melt it into a data.frame:



```
a <- array(c(1:23, NA), c(2,3,4))
data.frame(melt(a))
```

In Python, since `a` is a list, you can simply use list comprehension.

```
In [28]: a = np.array(list(range(1,24))+[np.NaN]).reshape(2,3,4)

In [29]: pd.DataFrame([tuple(list(x)+[val]) for x, val in np.ndenumerate(a)])
Out[29]:
```

	0	1	2	3
0	0	0	0	1.0
1	0	0	1	2.0
2	0	0	2	3.0
3	0	0	3	4.0
4	0	1	0	5.0
5	0	1	1	6.0
6	0	1	2	7.0
..	..	..	..	...
17	1	1	1	18.0
18	1	1	2	19.0
19	1	1	3	20.0
20	1	2	0	21.0
21	1	2	1	22.0
22	1	2	2	23.0
23	1	2	3	NaN

[24 rows x 4 columns]

### 30.4.2 melt.list

An expression using a list called `a` in R where you want to melt it into a data.frame:

```
a <- as.list(c(1:4, NA))
data.frame(melt(a))
```

In Python, this list would be a list of tuples, so `DataFrame()` method would convert it to a dataframe as required.

```
In [30]: a = list(enumerate(list(range(1,5))+[np.NaN]))

In [31]: pd.DataFrame(a)
Out[31]:
```

	0	1
0	0	1.0
1	1	2.0
2	2	3.0
3	3	4.0
4	4	NaN

For more details and examples see [the Into to Data Structures documentation](#).

### 30.4.3 melt.data.frame

An expression using a data.frame called `cheese` in R where you want to reshape the data.frame:

```
cheese <- data.frame(
  first = c('John', 'Mary'),
  last = c('Doe', 'Bo'),
  height = c(5.5, 6.0),
  weight = c(130, 150)
)
melt(cheese, id=c("first", "last"))
```

In Python, the `melt()` method is the R equivalent:

```
In [32]: cheese = pd.DataFrame({'first' : ['John', 'Mary'],
.....:                          'last' : ['Doe', 'Bo'],
.....:                          'height' : [5.5, 6.0],
.....:                          'weight' : [130, 150]})
.....:
```

```
In [33]: pd.melt(cheese, id_vars=['first', 'last'])
```

```
Out[33]:
```

	first	last	variable	value
0	John	Doe	height	5.5
1	Mary	Bo	height	6.0
2	John	Doe	weight	130.0
3	Mary	Bo	weight	150.0

```
In [34]: cheese.set_index(['first', 'last']).stack() # alternative way
//////////
↪
first  last
John   Doe    height      5.5
        weight    130.0
Mary   Bo     height      6.0
        weight    150.0
dtype: float64
```

For more details and examples see *the reshaping documentation*.

### 30.4.4 cast

In R `acast` is an expression using a `data.frame` called `df` in R to cast into a higher dimensional array:

```
df <- data.frame(
  x = runif(12, 1, 168),
  y = runif(12, 7, 334),
  z = runif(12, 1.7, 20.7),
  month = rep(c(5,6,7), 4),
  week = rep(c(1,2), 6)
)

mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

In Python the best way is to make use of `pivot_table()`:

```
In [35]: df = pd.DataFrame({
.....:     'x': np.random.uniform(1., 168., 12),
.....:     'y': np.random.uniform(7., 334., 12),
```

(continues on next page)

(continued from previous page)

```

.....:     'z': np.random.uniform(1.7, 20.7, 12),
.....:     'month': [5,6,7]*4,
.....:     'week': [1,2]*6
.....: })
.....:

In [36]: mdf = pd.melt(df, id_vars=['month', 'week'])

In [37]: pd.pivot_table(mdf, values='value', index=['variable', 'week'],
.....:                  columns=['month'], aggfunc=np.mean)
.....:
Out[37]:

```

		5	6	7
variable	week			
x	1	114.001700	132.227290	65.808204
	2	124.669553	147.495706	82.882820
y	1	225.636630	301.864228	91.706834
	2	57.692665	215.851669	218.004383
z	1	17.793871	7.124644	17.679823
	2	15.068355	13.873974	9.394966

Similarly for `dcast` which uses a data.frame called `df` in R to aggregate information based on `Animal` and `FeedType`:

```

df <- data.frame(
  Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
            'Animal2', 'Animal3'),
  FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
  Amount = c(10, 7, 4, 2, 5, 6, 2)
)

dcast(df, Animal ~ FeedType, sum, fill=NaN)
# Alternative method using base R
with(df, tapply(Amount, list(Animal, FeedType), sum))

```

Python can approach this in two different ways. Firstly, similar to above using `pivot_table()`:

```

In [38]: df = pd.DataFrame({
.....:     'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
.....:               'Animal2', 'Animal3'],
.....:     'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
.....:     'Amount': [10, 7, 4, 2, 5, 6, 2],
.....: })
.....:

In [39]: df.pivot_table(values='Amount', index='Animal', columns='FeedType', aggfunc=
↪ 'sum')
Out[39]:

```

	A	B
Animal		
Animal1	10.0	5.0
Animal2	2.0	13.0
Animal3	6.0	NaN

The second approach is to use the `groupby()` method:

```
In [40]: df.groupby(['Animal', 'FeedType'])['Amount'].sum()
Out[40]:
```

Animal	FeedType	Amount
Animal1	A	10
	B	5
Animal2	A	2
	B	13
Animal3	A	6

Name: Amount, dtype: int64

For more details and examples see [the reshaping documentation](#) or [the groupby documentation](#).

### 30.4.5 factor

pandas has a data type for categorical data.

```
cut(c(1, 2, 3, 4, 5, 6), 3)
factor(c(1, 2, 3, 2, 2, 3))
```

In pandas this is accomplished with `pd.cut` and `astype("category")`:

```
In [41]: pd.cut(pd.Series([1,2,3,4,5,6]), 3)
Out[41]:
0      (0.995, 2.667]
1      (0.995, 2.667]
2      (2.667, 4.333]
3      (2.667, 4.333]
4      (4.333, 6.0]
5      (4.333, 6.0]
dtype: category
Categories (3, interval[float64]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.0]]

In [42]: pd.Series([1,2,3,2,2,3]).astype("category")
=====
↪
0      1
1      2
2      3
3      2
4      2
5      3
dtype: category
Categories (3, int64): [1, 2, 3]
```

For more details and examples see *categorical introduction* and the *API documentation*. There is also a documentation regarding the *differences to R's factor*.

## COMPARISON WITH SQL

Since many potential pandas users have some familiarity with [SQL](#), this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd
```

```
In [2]: import numpy as np
```

Most of the examples will utilize the `tips` dataset found within pandas tests. We'll read the data into a DataFrame called `tips` and assume we have a database table of the same name and structure.

```
In [3]: url = 'https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/tips.
↳ csv'
```

```
In [4]: tips = pd.read_csv(url)
```

```
In [5]: tips.head()
```

```
Out[5]:
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

### 31.1 SELECT

In SQL, selection is done using a comma-separated list of columns you'd like to select (or a `*` to select all columns):

```
SELECT total_bill, tip, smoker, time
FROM tips
LIMIT 5;
```

With pandas, column selection is done by passing a list of column names to your DataFrame:

```
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
```

```
Out[6]:
```

	total_bill	tip	smoker	time
--	------------	-----	--------	------

(continues on next page)

(continued from previous page)

0	16.99	1.01	No	Dinner
1	10.34	1.66	No	Dinner
2	21.01	3.50	No	Dinner
3	23.68	3.31	No	Dinner
4	24.59	3.61	No	Dinner

Calling the DataFrame without the list of column names would display all columns (akin to SQL's \*).

## 31.2 WHERE

Filtering in SQL is done via a WHERE clause.

```
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using [boolean indexing](#).

```
In [7]: tips[tips['time'] == 'Dinner'].head(5)
Out[7]:
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

The above statement is simply passing a Series of True/False objects to the DataFrame, returning all rows with True.

```
In [8]: is_dinner = tips['time'] == 'Dinner'

In [9]: is_dinner.value_counts()
Out[9]:
```

True	176
False	68

```
Name: time, dtype: int64

In [10]: tips[is_dinner].head(5)
Out[10]:
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Just like SQL's OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

```
-- tips of more than $5.00 at Dinner meals
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;
```

```
# tips of more than $5.00 at Dinner meals
In [11]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
```

```
Out[11]:
```

	total_bill	tip	sex	smoker	day	time	size
23	39.42	7.58	Male	No	Sat	Dinner	4
44	30.40	5.60	Male	No	Sun	Dinner	4
47	32.40	6.00	Male	No	Sun	Dinner	4
52	34.81	5.20	Female	No	Sun	Dinner	4
59	48.27	6.73	Male	No	Sat	Dinner	4
116	29.93	5.07	Male	No	Sun	Dinner	4
155	29.85	5.14	Female	No	Sun	Dinner	5
170	50.81	10.00	Male	Yes	Sat	Dinner	3
172	7.25	5.15	Male	Yes	Sun	Dinner	2
181	23.33	5.65	Male	Yes	Sun	Dinner	2
183	23.17	6.50	Male	Yes	Sun	Dinner	4
211	25.89	5.16	Male	Yes	Sat	Dinner	4
212	48.33	9.00	Male	No	Sat	Dinner	4
214	28.17	6.50	Female	Yes	Sat	Dinner	3
239	29.03	5.92	Male	No	Sat	Dinner	3

```
-- tips by parties of at least 5 diners OR bill total was more than $45
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;
```

```
# tips by parties of at least 5 diners OR bill total was more than $45
```

```
In [12]: tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
```

```
Out[12]:
```

	total_bill	tip	sex	smoker	day	time	size
59	48.27	6.73	Male	No	Sat	Dinner	4
125	29.80	4.20	Female	No	Thur	Lunch	6
141	34.30	6.70	Male	No	Thur	Lunch	6
142	41.19	5.00	Male	No	Thur	Lunch	5
143	27.05	5.00	Female	No	Thur	Lunch	6
155	29.85	5.14	Female	No	Sun	Dinner	5
156	48.17	5.00	Male	No	Sun	Dinner	6
170	50.81	10.00	Male	Yes	Sat	Dinner	3
182	45.35	3.50	Male	Yes	Sun	Dinner	3
185	20.69	5.00	Male	No	Sun	Dinner	5
187	30.46	2.00	Male	Yes	Sun	Dinner	5
212	48.33	9.00	Male	No	Sat	Dinner	4
216	28.15	3.00	Male	Yes	Sat	Dinner	5

NULL checking is done using the `notna()` and `isna()` methods.

```
In [13]: frame = pd.DataFrame({'col1': ['A', 'B', np.NaN, 'C', 'D'],
.....:                        'col2': ['F', np.NaN, 'G', 'H', 'I']})
.....:
```

```
In [14]: frame
```

```
Out[14]:
```

	col1	col2
0	A	F
1	B	NaN
2	NaN	G
3	C	H
4	D	I

Assume we have a table of the same structure as our DataFrame above. We can see only the records where `col2` IS NULL with the following query:

```
SELECT *
FROM frame
WHERE col2 IS NULL;
```

```
In [15]: frame[frame['col2'].isna()]
Out[15]:
   col1 col2
1     B  NaN
```

Getting items where `col1` IS NOT NULL can be done with `notna()`.

```
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
```

```
In [16]: frame[frame['col1'].notna()]
Out[16]:
   col1 col2
0     A     F
1     B  NaN
3     C     H
4     D     I
```

## 31.3 GROUP BY

In pandas, SQL's GROUP BY operations are performed using the similarly named `groupby()` method. `groupby()` typically refers to a process where we'd like to split a dataset into groups, apply some function (typically aggregation), and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

```
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female      87
Male       157
*/
```

The pandas equivalent would be:

```
In [17]: tips.groupby('sex').size()
Out[17]:
sex
Female      87
Male       157
dtype: int64
```

Notice that in the pandas code we used `size()` and not `count()`. This is because `count()` applies the function to each column, returning the number of not null records within each.



```
In [18]: tips.groupby('sex').count()
Out[18]:
```

	total_bill	tip	smoker	day	time	size
sex						
Female	87	87	87	87	87	87
Male	157	157	157	157	157	157

Alternatively, we could have applied the `count()` method to an individual column:

```
In [19]: tips.groupby('sex')['total_bill'].count()
Out[19]:
```

sex	total_bill
Female	87
Male	157

Name: total\_bill, dtype: int64

Multiple functions can also be applied at once. For instance, say we'd like to see how tip amount differs by day of the week - `agg()` allows you to pass a dictionary to your grouped DataFrame, indicating which functions to apply to specific columns.

```
SELECT day, AVG(tip), COUNT(*)
FROM tips
GROUP BY day;
/*
Fri    2.734737    19
Sat    2.993103    87
Sun    3.255132    76
Thur   2.771452    62
*/
```

```
In [20]: tips.groupby('day').agg({'tip': np.mean, 'day': np.size})
Out[20]:
```

	tip	day
day		
Fri	2.734737	19
Sat	2.993103	87
Sun	3.255132	76
Thur	2.771452	62

Grouping by more than one column is done by passing a list of columns to the `groupby()` method.

```
SELECT smoker, day, COUNT(*), AVG(tip)
FROM tips
GROUP BY smoker, day;
/*
smoker day
No      Fri      4  2.812500
        Sat     45  3.102889
        Sun     57  3.167895
        Thur    45  2.673778
Yes     Fri     15  2.714000
        Sat     42  2.875476
        Sun     19  3.516842
        Thur    17  3.030000
*/
```

```
In [21]: tips.groupby(['smoker', 'day']).agg({'tip': [np.size, np.mean]})
Out[21]:
```

		tip	
		size	mean
smoker	day		
No	Fri	4.0	2.812500
	Sat	45.0	3.102889
	Sun	57.0	3.167895
	Thur	45.0	2.673778
Yes	Fri	15.0	2.714000
	Sat	42.0	2.875476
	Sun	19.0	3.516842
	Thur	17.0	3.030000

## 31.4 JOIN

JOINS can be performed with `join()` or `merge()`. By default, `join()` will join the DataFrames on their indices. Each method has parameters allowing you to specify the type of join to perform (LEFT, RIGHT, INNER, FULL) or the columns to join on (column names or indices).

```
In [22]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
.....:                      'value': np.random.randn(4)})
.....:

In [23]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
.....:                      'value': np.random.randn(4)})
.....:
```

Assume we have two database tables of the same name and structure as our DataFrames.

Now let's go over the various types of JOINS.

### 31.4.1 INNER JOIN

```
SELECT *
FROM df1
INNER JOIN df2
ON df1.key = df2.key;
```

```
# merge performs an INNER JOIN by default
```

```
In [24]: pd.merge(df1, df2, on='key')
```

```
Out[24]:
   key  value_x  value_y
0    B -0.318214  0.543581
1    D  2.169960 -0.426067
2    D  2.169960  1.138079
```

`merge()` also offers parameters for cases when you'd like to join one DataFrame's column with another DataFrame's index.

```
In [25]: indexed_df2 = df2.set_index('key')
```

```
In [26]: pd.merge(df1, indexed_df2, left_on='key', right_index=True)
```

(continues on next page)

(continued from previous page)

**Out [26]:**

	key	value_x	value_y
1	B	-0.318214	0.543581
3	D	2.169960	-0.426067
3	D	2.169960	1.138079

### 31.4.2 LEFT OUTER JOIN

```
-- show all records from df1
SELECT *
FROM df1
LEFT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from df1
In [27]: pd.merge(df1, df2, on='key', how='left')
Out [27]:
```

	key	value_x	value_y
0	A	0.116174	NaN
1	B	-0.318214	0.543581
2	C	0.285261	NaN
3	D	2.169960	-0.426067
4	D	2.169960	1.138079

### 31.4.3 RIGHT JOIN

```
-- show all records from df2
SELECT *
FROM df1
RIGHT OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from df2
In [28]: pd.merge(df1, df2, on='key', how='right')
Out [28]:
```

	key	value_x	value_y
0	B	-0.318214	0.543581
1	D	2.169960	-0.426067
2	D	2.169960	1.138079
3	E	NaN	0.086073

### 31.4.4 FULL JOIN

pandas also allows for FULL JOINS, which display both sides of the dataset, whether or not the joined columns find a match. As of writing, FULL JOINS are not supported in all RDBMS (MySQL).

```
-- show all records from both tables
SELECT *
FROM df1
FULL OUTER JOIN df2
  ON df1.key = df2.key;
```

```
# show all records from both frames
In [29]: pd.merge(df1, df2, on='key', how='outer')
Out[29]:
```

	key	value_x	value_y
0	A	0.116174	NaN
1	B	-0.318214	0.543581
2	C	0.285261	NaN
3	D	2.169960	-0.426067
4	D	2.169960	1.138079
5	E	NaN	0.086073

## 31.5 UNION

UNION ALL can be performed using `concat()`.

```
In [30]: df1 = pd.DataFrame({'city': ['Chicago', 'San Francisco', 'New York City'],
.....:                      'rank': range(1, 4)})
.....:

In [31]: df2 = pd.DataFrame({'city': ['Chicago', 'Boston', 'Los Angeles'],
.....:                      'rank': [1, 4, 5]})
.....:
```

```
SELECT city, rank
FROM df1
UNION ALL
SELECT city, rank
FROM df2;
/*
      city  rank
0  Chicago    1
1 San Francisco  2
2 New York City  3
0  Chicago    1
1   Boston    4
2 Los Angeles  5
*/
```

```
In [32]: pd.concat([df1, df2])
Out[32]:
```

	city	rank
0	Chicago	1
1	San Francisco	2
2	New York City	3
0	Chicago	1
1	Boston	4
2	Los Angeles	5

SQL's UNION is similar to UNION ALL, however UNION will remove duplicate rows.

```
SELECT city, rank
FROM df1
UNION
SELECT city, rank
```

(continues on next page)

(continued from previous page)

```

FROM df2;
-- notice that there is only one Chicago record this time
/*
      city  rank
    Chicago    1
San Francisco    2
New York City    3
      Boston    4
    Los Angeles    5
*/

```

In pandas, you can use `concat()` in conjunction with `drop_duplicates()`.

```

In [33]: pd.concat([df1, df2]).drop_duplicates()
Out[33]:
      city  rank
0   Chicago    1
1 San Francisco    2
2 New York City    3
1    Boston    4
2  Los Angeles    5

```

## 31.6 Pandas equivalents for some SQL analytic and aggregate functions

### 31.6.1 Top N rows with offset

```

-- MySQL
SELECT * FROM tips
ORDER BY tip DESC
LIMIT 10 OFFSET 5;

```

```

In [34]: tips.nlargest(10+5, columns='tip').tail(10)
Out[34]:
   total_bill  tip  sex smoker  day  time  size
183      23.17  6.50  Male   Yes  Sun  Dinner    4
214      28.17  6.50  Female Yes  Sat  Dinner    3
47       32.40  6.00  Male   No  Sun  Dinner    4
239      29.03  5.92  Male   No  Sat  Dinner    3
88       24.71  5.85  Male   No  Thur  Lunch    2
181      23.33  5.65  Male  Yes  Sun  Dinner    2
44       30.40  5.60  Male   No  Sun  Dinner    4
52       34.81  5.20  Female  No  Sun  Dinner    4
85       34.83  5.17  Female  No  Thur  Lunch    4
211      25.89  5.16  Male  Yes  Sat  Dinner    4

```

### 31.6.2 Top N rows per group

```

-- Oracle's ROW_NUMBER() analytic function
SELECT * FROM (

```

(continues on next page)

(continued from previous page)

```

SELECT
    t.*,
    ROW_NUMBER() OVER(PARTITION BY day ORDER BY total_bill DESC) AS rn
FROM tips t
)
WHERE rn < 3
ORDER BY day, rn;

```

```

In [35]: (tips.assign(rn=tips.sort_values(['total_bill'], ascending=False)
.....:               .groupby(['day'])
.....:               .cumcount() + 1)
.....:   .query('rn < 3')
.....:   .sort_values(['day', 'rn'])
.....: )
.....:
Out[35]:

```

	total_bill	tip	sex	smoker	day	time	size	rn
95	40.17	4.73	Male	Yes	Fri	Dinner	4	1
90	28.97	3.00	Male	Yes	Fri	Dinner	2	2
170	50.81	10.00	Male	Yes	Sat	Dinner	3	1
212	48.33	9.00	Male	No	Sat	Dinner	4	2
156	48.17	5.00	Male	No	Sun	Dinner	6	1
182	45.35	3.50	Male	Yes	Sun	Dinner	3	2
197	43.11	5.00	Female	Yes	Thur	Lunch	4	1
142	41.19	5.00	Male	No	Thur	Lunch	5	2

the same using `rank(method='first')` function

```

In [36]: (tips.assign(rnk=tips.groupby(['day'])['total_bill']
.....:               .rank(method='first', ascending=False))
.....:   .query('rnk < 3')
.....:   .sort_values(['day', 'rnk'])
.....: )
.....:
Out[36]:

```

	total_bill	tip	sex	smoker	day	time	size	rnk
95	40.17	4.73	Male	Yes	Fri	Dinner	4	1.0
90	28.97	3.00	Male	Yes	Fri	Dinner	2	2.0
170	50.81	10.00	Male	Yes	Sat	Dinner	3	1.0
212	48.33	9.00	Male	No	Sat	Dinner	4	2.0
156	48.17	5.00	Male	No	Sun	Dinner	6	1.0
182	45.35	3.50	Male	Yes	Sun	Dinner	3	2.0
197	43.11	5.00	Female	Yes	Thur	Lunch	4	1.0
142	41.19	5.00	Male	No	Thur	Lunch	5	2.0

```

-- Oracle's RANK() analytic function
SELECT * FROM (
    SELECT
        t.*,
        RANK() OVER(PARTITION BY sex ORDER BY tip) AS rnk
    FROM tips t
    WHERE tip < 2
)
WHERE rnk < 3
ORDER BY sex, rnk;

```

Let's find tips with (rank < 3) per gender group for (tips < 2). Notice that when using `rank(method='min')`

function *rnk\_min* remains the same for the same *tip* (as Oracle's RANK() function)

```
In [37]: (tips[tips['tip'] < 2]
....:      .assign(rnk_min=tips.groupby(['sex'])['tip']
....:              .rank(method='min'))
....:      .query('rnk_min < 3')
....:      .sort_values(['sex', 'rnk_min'])
....: )
....:
Out[37]:
```

	total_bill	tip	sex	smoker	day	time	size	rnk_min
67	3.07	1.00	Female	Yes	Sat	Dinner	1	1.0
92	5.75	1.00	Female	Yes	Fri	Dinner	2	1.0
111	7.25	1.00	Female	No	Sat	Dinner	1	1.0
236	12.60	1.00	Male	Yes	Sat	Dinner	2	1.0
237	32.83	1.17	Male	Yes	Sat	Dinner	2	2.0

## 31.7 UPDATE

```
UPDATE tips
SET tip = tip*2
WHERE tip < 2;
```

```
In [38]: tips.loc[tips['tip'] < 2, 'tip'] *= 2
```

## 31.8 DELETE

```
DELETE FROM tips
WHERE tip > 9;
```

In pandas we select the rows that should remain, instead of deleting them

```
In [39]: tips = tips.loc[tips['tip'] <= 9]
```





## COMPARISON WITH SAS

For potential users coming from SAS this page is meant to demonstrate how different SAS operations would be performed in pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

```
In [1]: import pandas as pd
In [2]: import numpy as np
```

**Note:** Throughout this tutorial, the pandas `DataFrame` will be displayed by calling `df.head()`, which displays the first N (default 5) rows of the `DataFrame`. This is often used in interactive work (e.g. [Jupyter notebook](#) or terminal) - the equivalent in SAS would be:

```
proc print data=df(obs=5);
run;
```

### 32.1 Data Structures

#### 32.1.1 General Terminology Translation

pandas	SAS
<code>DataFrame</code>	data set
column	variable
row	observation
<code>groupby</code>	BY-group
NaN	.

#### 32.1.2 `DataFrame` / `Series`

A `DataFrame` in pandas is analogous to a SAS data set - a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set using SAS's `DATA` step, can also be accomplished in pandas.

A `Series` is the data structure that represents one column of a `DataFrame`. SAS doesn't have a separate data structure for a single column, but in general, working with a `Series` is analogous to referencing a column in the `DATA` step.

### 32.1.3 Index

Every `DataFrame` and `Series` has an `Index` - which are labels on the *rows* of the data. SAS does not have an exactly analogous concept. A data set's rows are essentially unlabeled, other than an implicit integer index that can be accessed during the `DATA` step (`_N_`).

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled `Index` or `MultiIndex` can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the `Index` and just treat the `DataFrame` as a collection of columns. Please see the [indexing documentation](#) for much more on how to use an `Index` effectively.

## 32.2 Data Input / Output

### 32.2.1 Constructing a DataFrame from Values

A SAS data set can be built from specified values by placing the data after a `datalines` statement and specifying the column names.

```
data df;
  input x y;
  datalines;
1 2
3 4
5 6
;
run;
```

A pandas `DataFrame` can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({
...:     'x': [1, 3, 5],
...:     'y': [2, 4, 6]})
...:

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

### 32.2.2 Reading External Data

Like SAS, pandas provides utilities for reading in data from many formats. The `tips` dataset, found within the pandas tests (`csv`) will be used in many of the following examples.

SAS provides `PROC IMPORT` to read csv data into a data set.

```
proc import datafile='tips.csv' dbms=csv out=tips replace;
    getnames=yes;
run;
```

The pandas method is `read_csv()`, which works similarly.

```
In [5]: url = 'https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/tips.
↳ csv'

In [6]: tips = pd.read_csv(url)

In [7]: tips.head()
Out[7]:
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Like PROC IMPORT, `read_csv` can take a number of parameters to specify how the data should be parsed. For example, if the data was instead tab delimited, and did not have column names, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

In addition to text/csv, pandas supports a variety of other data formats such as Excel, HDF5, and SQL databases. These are all read via a `pd.read_*` function. See the [IO documentation](#) for more details.

### 32.2.3 Exporting Data

The inverse of PROC IMPORT in SAS is PROC EXPORT

```
proc export data=tips outfile='tips2.csv' dbms=csv;
run;
```

Similarly in pandas, the opposite of `read_csv` is `to_csv()`, and other data formats follow a similar api.

```
tips.to_csv('tips2.csv')
```

## 32.3 Data Operations

### 32.3.1 Operations on Columns

In the DATA step, arbitrary math expressions can be used on new or existing columns.

```
data tips;
    set tips;
    total_bill = total_bill - 2;
    new_bill = total_bill / 2;
run;
```

pandas provides similar vectorized operations by specifying the individual Series in the DataFrame. New columns can be assigned in the same way.

```
In [8]: tips['total_bill'] = tips['total_bill'] - 2
In [9]: tips['new_bill'] = tips['total_bill'] / 2.0
In [10]: tips.head()
Out[10]:
```

	total_bill	tip	sex	smoker	day	time	size	new_bill
0	14.99	1.01	Female	No	Sun	Dinner	2	7.495
1	8.34	1.66	Male	No	Sun	Dinner	3	4.170
2	19.01	3.50	Male	No	Sun	Dinner	3	9.505
3	21.68	3.31	Male	No	Sun	Dinner	2	10.840
4	22.59	3.61	Female	No	Sun	Dinner	4	11.295

### 32.3.2 Filtering

Filtering in SAS is done with an `if` or `where` statement, on one or more columns.

```
data tips;
  set tips;
  if total_bill > 10;
run;

data tips;
  set tips;
  where total_bill > 10;
  /* equivalent in this case - where happens before the
     DATA step begins and can also be used in PROC statements */
run;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*

```
In [11]: tips[tips['total_bill'] > 10].head()
Out[11]:
```

	total_bill	tip	sex	smoker	day	time	size
0	14.99	1.01	Female	No	Sun	Dinner	2
2	19.01	3.50	Male	No	Sun	Dinner	3
3	21.68	3.31	Male	No	Sun	Dinner	2
4	22.59	3.61	Female	No	Sun	Dinner	4
5	23.29	4.71	Male	No	Sun	Dinner	4

### 32.3.3 If/Then Logic

In SAS, if/then logic can be used to create new columns.

```
data tips;
  set tips;
  format bucket $4.;

  if total_bill < 10 then bucket = 'low';
  else bucket = 'high';
run;
```

The same operation in pandas can be accomplished using the `where` method from `numpy`.

```
In [12]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [13]: tips.head()
Out[13]:
```

	total_bill	tip	sex	smoker	day	time	size	bucket
0	14.99	1.01	Female	No	Sun	Dinner	2	high
1	8.34	1.66	Male	No	Sun	Dinner	3	low
2	19.01	3.50	Male	No	Sun	Dinner	3	high
3	21.68	3.31	Male	No	Sun	Dinner	2	high
4	22.59	3.61	Female	No	Sun	Dinner	4	high

### 32.3.4 Date Functionality

SAS provides a variety of functions to do operations on date/datetime columns.

```
data tips;
  set tips;
  format date1 date2 date1_plusmonth mmddyy10.;
  date1 = mdy(1, 15, 2013);
  date2 = mdy(2, 15, 2015);
  date1_year = year(date1);
  date2_month = month(date2);
  * shift date to beginning of next interval;
  date1_next = intnx('MONTH', date1, 1);
  * count intervals between dates;
  months_between = intck('MONTH', date1, date2);
run;
```

The equivalent pandas operations are shown below. In addition to these functions pandas supports other Time Series features not available in Base SAS (such as resampling and custom offsets) - see the [timeseries documentation](#) for more details.

```
In [14]: tips['date1'] = pd.Timestamp('2013-01-15')

In [15]: tips['date2'] = pd.Timestamp('2015-02-15')

In [16]: tips['date1_year'] = tips['date1'].dt.year

In [17]: tips['date2_month'] = tips['date2'].dt.month

In [18]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [19]: tips['months_between'] = (tips['date2'].dt.to_period('M') -
....:                               tips['date1'].dt.to_period('M'))
....:

In [20]: tips[['date1', 'date2', 'date1_year', 'date2_month',
....:           'date1_next', 'months_between']].head()
....:
```

	date1	date2	date1_year	date2_month	date1_next	months_between
0	2013-01-15	2015-02-15	2013	2	2013-02-01	25
1	2013-01-15	2015-02-15	2013	2	2013-02-01	25
2	2013-01-15	2015-02-15	2013	2	2013-02-01	25

(continues on next page)

(continued from previous page)

3	2013-01-15	2015-02-15	2013	2	2013-02-01	25
4	2013-01-15	2015-02-15	2013	2	2013-02-01	25

### 32.3.5 Selection of Columns

SAS provides keywords in the DATA step to select, drop, and rename columns.

```
data tips;
    set tips;
    keep sex total_bill tip;
run;

data tips;
    set tips;
    drop sex;
run;

data tips;
    set tips;
    rename total_bill=total_bill_2;
run;
```

The same operations are expressed in pandas below.

```
# keep
In [21]: tips[['sex', 'total_bill', 'tip']].head()
Out[21]:
```

	sex	total_bill	tip
0	Female	14.99	1.01
1	Male	8.34	1.66
2	Male	19.01	3.50
3	Male	21.68	3.31
4	Female	22.59	3.61

```
# drop
In [22]: tips.drop('sex', axis=1).head()
```

```
Out[22]:
```

	total_bill	tip	smoker	day	time	size
0	14.99	1.01	No	Sun	Dinner	2
1	8.34	1.66	No	Sun	Dinner	3
2	19.01	3.50	No	Sun	Dinner	3
3	21.68	3.31	No	Sun	Dinner	2
4	22.59	3.61	No	Sun	Dinner	4

```
# rename
In [23]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
```

```
Out[23]:
```

	total_bill_2	tip	sex	smoker	day	time	size
0	14.99	1.01	Female	No	Sun	Dinner	2
1	8.34	1.66	Male	No	Sun	Dinner	3
2	19.01	3.50	Male	No	Sun	Dinner	3
3	21.68	3.31	Male	No	Sun	Dinner	2
4	22.59	3.61	Female	No	Sun	Dinner	4

### 32.3.6 Sorting by Values

Sorting in SAS is accomplished via PROC SORT

```
proc sort data=tips;
    by sex total_bill;
run;
```

pandas objects have a `sort_values()` method, which takes a list of columns to sort by.

```
In [24]: tips = tips.sort_values(['sex', 'total_bill'])

In [25]: tips.head()
Out[25]:
```

	total_bill	tip	sex	smoker	day	time	size
67	1.07	1.00	Female	Yes	Sat	Dinner	1
92	3.75	1.00	Female	Yes	Fri	Dinner	2
111	5.25	1.00	Female	No	Sat	Dinner	1
145	6.35	1.50	Female	No	Thur	Lunch	2
135	6.51	1.25	Female	No	Thur	Lunch	2

## 32.4 String Processing

### 32.4.1 Length

SAS determines the length of a character string with the `LENGTHN` and `LENGTHC` functions. `LENGTHN` excludes trailing blanks and `LENGTHC` includes trailing blanks.

```
data _null_;
set tips;
put (LENGTHN(time));
put (LENGTHC(time));
run;
```

Python determines the length of a character string with the `len` function. `len` includes trailing blanks. Use `len` and `rstrip` to exclude trailing blanks.

```
In [26]: tips['time'].str.len().head()
Out[26]:
```

67	6
92	6
111	6
145	5
135	5

Name: time, dtype: int64

```
In [27]: tips['time'].str.rstrip().str.len().head()
Out[27]:
```

67	6
92	6
111	6
145	5
135	5

Name: time, dtype: int64

### 32.4.2 Find

SAS determines the position of a character in a string with the **FINDW** function. **FINDW** takes the string defined by the first argument and searches for the first position of the substring you supply as the second argument.

```
data _null_;
set tips;
put (FINDW(sex, 'ale'));
run;
```

Python determines the position of a character in a string with the `find` function. `find` searches for the first position of the substring. If the substring is found, the function returns its position. Keep in mind that Python indexes are zero-based and the function will return -1 if it fails to find the substring.

```
In [28]: tips['sex'].str.find("ale").head()
Out[28]:
67      3
92      3
111     3
145     3
135     3
Name: sex, dtype: int64
```

### 32.4.3 Substring

SAS extracts a substring from a string based on its position with the **SUBSTR** function.

```
data _null_;
set tips;
put (substr(sex, 1, 1));
run;
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [29]: tips['sex'].str[0:1].head()
Out[29]:
67      F
92      F
111     F
145     F
135     F
Name: sex, dtype: object
```

### 32.4.4 Scan

The SAS **SCAN** function returns the *n*th word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
data firstlast;
input String $60.;
First_Name = scan(string, 1);
Last_Name = scan(string, -1);
datalines2;
```

(continues on next page)



(continued from previous page)

```
John Smith;
Jane Cook;
;;;
run;
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [30]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})

In [31]: firstlast['First_Name'] = firstlast['String'].str.split(" ", expand=True)[0]

In [32]: firstlast['Last_Name'] = firstlast['String'].str.rsplit(" ", expand=True)[0]

In [33]: firstlast
Out[33]:
```

	String	First_Name	Last_Name
0	John Smith	John	Smith
1	Jane Cook	Jane	Cook

### 32.4.5 Uppcase, Lowcase, and Propcase

The SAS `UPCASE` `LOWCASE` and `PROPCASE` functions change the case of the argument.

```
data firstlast;
input String $60.;
string_up = UPCASE(string);
string_low = LOWCASE(string);
string_prop = PROPCASE(string);
datalines2;
John Smith;
Jane Cook;
;;;
run;
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [34]: firstlast = pd.DataFrame({'String': ['John Smith', 'Jane Cook']})

In [35]: firstlast['string_up'] = firstlast['String'].str.upper()

In [36]: firstlast['string_low'] = firstlast['String'].str.lower()

In [37]: firstlast['string_prop'] = firstlast['String'].str.title()

In [38]: firstlast
Out[38]:
```

	String	string_up	string_low	string_prop
0	John Smith	JOHN SMITH	john smith	John Smith
1	Jane Cook	JANE COOK	jane cook	Jane Cook

## 32.5 Merging

The following tables will be used in the merge examples

```
In [39]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
.....:                      'value': np.random.randn(4)})
.....:

In [40]: df1
Out[40]:
   key  value
0    A -0.857326
1    B  1.075416
2    C  0.371727
3    D  1.065735

In [41]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
.....:                      'value': np.random.randn(4)})
.....:

In [42]: df2
Out[42]:
   key  value
0    B -0.227314
1    D  2.102726
2    D -0.092796
3    E  0.094694
```

In SAS, data must be explicitly sorted before merging. Different types of joins are accomplished using the `in=` dummy variables to track whether a match was found in one or both input frames.

```
proc sort data=df1;
  by key;
run;

proc sort data=df2;
  by key;
run;

data left_join inner_join right_join outer_join;
  merge df1(in=a) df2(in=b);

  if a and b then output inner_join;
  if a then output left_join;
  if b then output right_join;
  if a or b then output outer_join;
run;
```

pandas DataFrames have a `merge()` method, which provides similar functionality. Note that the data does not have to be sorted ahead of time, and different join types are accomplished via the `how` keyword.

```
In [43]: inner_join = df1.merge(df2, on='key', how='inner')

In [44]: inner_join
Out[44]:
   key  value_x  value_y
0    B  1.075416 -0.227314
```

(continues on next page)

(continued from previous page)

```
1  D  1.065735  2.102726
2  D  1.065735 -0.092796
```

```
In [45]: left_join = df1.merge(df2, on=['key'], how='left')
```

```
In [46]: left_join
```

```
Out[46]:
```

```
   key  value_x  value_y
0  A -0.857326      NaN
1  B  1.075416 -0.227314
2  C  0.371727      NaN
3  D  1.065735  2.102726
4  D  1.065735 -0.092796
```

```
In [47]: right_join = df1.merge(df2, on=['key'], how='right')
```

```
In [48]: right_join
```

```
Out[48]:
```

```
   key  value_x  value_y
0  B  1.075416 -0.227314
1  D  1.065735  2.102726
2  D  1.065735 -0.092796
3  E      NaN  0.094694
```

```
In [49]: outer_join = df1.merge(df2, on=['key'], how='outer')
```

```
In [50]: outer_join
```

```
Out[50]:
```

```
   key  value_x  value_y
0  A -0.857326      NaN
1  B  1.075416 -0.227314
2  C  0.371727      NaN
3  D  1.065735  2.102726
4  D  1.065735 -0.092796
5  E      NaN  0.094694
```

## 32.6 Missing Data

Like SAS, pandas has a representation for missing data - which is the special float value NaN (not a number). Many of the semantics are the same, for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [51]: outer_join
```

```
Out[51]:
```

```
   key  value_x  value_y
0  A -0.857326      NaN
1  B  1.075416 -0.227314
2  C  0.371727      NaN
3  D  1.065735  2.102726
4  D  1.065735 -0.092796
5  E      NaN  0.094694
```

```
In [52]: outer_join['value_x'] + outer_join['value_y']
```

```
////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```

0      NaN
1    0.848102
2      NaN
3    3.168461
4    0.972939
5      NaN
dtype: float64

```

```
In [53]: outer_join['value_x'].sum()
```

```

////////////////////////////////////
↪ 2.7212865354426201

```

One difference is that missing data cannot be compared to its sentinel value. For example, in SAS you could do this to filter missing values.

```

data outer_join_nulls;
  set outer_join;
  if value_x = .;
run;

data outer_join_no_nulls;
  set outer_join;
  if value_x ^= .;
run;

```

Which doesn't work in pandas. Instead, the `pd.isna` or `pd.notna` functions should be used for comparisons.

```
In [54]: outer_join[pd.isna(outer_join['value_x'])]
```

```
Out[54]:
```

```

   key  value_x  value_y
5    E      NaN  0.094694

```

```
In [55]: outer_join[pd.notna(outer_join['value_x'])]
```

```

////////////////////////////////////Out[55]:

```

```

   key  value_x  value_y
0    A -0.857326      NaN
1    B  1.075416 -0.227314
2    C  0.371727      NaN
3    D  1.065735  2.102726
4    D  1.065735 -0.092796

```

pandas also provides a variety of methods to work with missing data - some of which would be challenging to express in SAS. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the [missing data documentation](#) for more.

```
In [56]: outer_join.dropna()
```

```
Out[56]:
```

```

   key  value_x  value_y
1    B  1.075416 -0.227314
3    D  1.065735  2.102726
4    D  1.065735 -0.092796

```

```
In [57]: outer_join.fillna(method='ffill')
```

```

////////////////////////////////////
↪
   key  value_x  value_y

```

(continues on next page)

(continued from previous page)

```

0   A  -0.857326      NaN
1   B   1.075416  -0.227314
2   C   0.371727  -0.227314
3   D   1.065735   2.102726
4   D   1.065735  -0.092796
5   E   1.065735   0.094694

```

```
In [58]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
```

```

////////////////////////////////////

```

```

→
0   -0.857326
1    1.075416
2    0.371727
3    1.065735
4    1.065735
5    0.544257
Name: value_x, dtype: float64

```

## 32.7 GroupBy

### 32.7.1 Aggregation

SAS's PROC SUMMARY can be used to group by one or more key variables and compute aggregations on numeric columns.

```

proc summary data=tips nway;
  class sex smoker;
  var total_bill tip;
  output out=tips_summed sum=;
run;

```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the [groupby documentation](#) for more details and examples.

```
In [59]: tips_summed = tips.groupby(['sex', 'smoker'])['total_bill', 'tip'].sum()
```

```
In [60]: tips_summed.head()
```

```
Out [60]:
```

		total_bill	tip
sex	smoker		
Female	No	869.68	149.77
	Yes	527.27	96.74
Male	No	1725.75	302.00
	Yes	1217.07	183.07

### 32.7.2 Transformation

In SAS, if the group aggregations need to be used with the original frame, it must be merged back together. For example, to subtract the mean for each observation by smoker group.

```

proc summary data=tips missing nway;
  class smoker;
  var total_bill;
  output out=smoker_means mean(total_bill)=group_bill;
run;

proc sort data=tips;
  by smoker;
run;

data tips;
  merge tips(in=a) smoker_means(in=b);
  by smoker;
  adj_total_bill = total_bill - group_bill;
  if a and b;
run;

```

pandas groupby provides a transform mechanism that allows these type of operations to be succinctly expressed in one operation.

```

In [61]: gb = tips.groupby('smoker')['total_bill']

In [62]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [63]: tips.head()
Out[63]:

```

	total_bill	tip	sex	smoker	day	time	size	adj_total_bill
67	1.07	1.00	Female	Yes	Sat	Dinner	1	-17.686344
92	3.75	1.00	Female	Yes	Fri	Dinner	2	-15.006344
111	5.25	1.00	Female	No	Sat	Dinner	1	-11.938278
145	6.35	1.50	Female	No	Thur	Lunch	2	-10.838278
135	6.51	1.25	Female	No	Thur	Lunch	2	-10.678278

### 32.7.3 By Group Processing

In addition to aggregation, pandas groupby can be used to replicate most other by group processing from SAS. For example, this DATA step reads the data by sex/smoker group and filters to the first entry for each.

```

proc sort data=tips;
  by sex smoker;
run;

data tips_first;
  set tips;
  by sex smoker;
  if FIRST.sex or FIRST.smoker then output;
run;

```

In pandas this would be written as:

```

In [64]: tips.groupby(['sex', 'smoker']).first()
Out[64]:

```

	total_bill	tip	day	time	size	adj_total_bill
sex smoker						
Female No	5.25	1.00	Sat	Dinner	1	-11.938278

(continues on next page)

(continued from previous page)

	Yes	1.07	1.00	Sat	Dinner	1	-17.686344
Male	No	5.51	2.00	Thur	Lunch	2	-11.678278
	Yes	5.25	5.15	Sun	Dinner	2	-13.506344

## 32.8 Other Considerations

### 32.8.1 Disk vs Memory

pandas operates exclusively in memory, where a SAS data set exists on disk. This means that the size of data able to be loaded in pandas is limited by your machine's memory, but also that the operations on that data may be faster.

If out of core processing is needed, one possibility is the [dask.dataframe](#) library (currently in development) which provides a subset of pandas functionality for an on-disk `DataFrame`

### 32.8.2 Data Interop

pandas provides a `read_sas()` method that can read SAS data saved in the XPORT or SAS7BDAT binary format.

```
libname xportout xport 'transport-file.xpt';
data xportout.tips;
    set tips(rename=(total_bill=tbill));
    * xport variable names limited to 6 characters;
run;
```

```
df = pd.read_sas('transport-file.xpt')
df = pd.read_sas('binary-file.sas7bdat')
```

You can also specify the file format directly. By default, pandas will try to infer the file format based on its extension.

```
df = pd.read_sas('transport-file.xpt', format='xport')
df = pd.read_sas('binary-file.sas7bdat', format='sas7bdat')
```

XPORT is a relatively limited format and the parsing of it is not as optimized as some of the other pandas readers. An alternative way to interop data between SAS and pandas is to serialize to csv.

```
# version 0.17, 10M rows

In [8]: %time df = pd.read_sas('big.xpt')
Wall time: 14.6 s

In [9]: %time df = pd.read_csv('big.csv')
Wall time: 4.86 s
```





## COMPARISON WITH STATA

For potential users coming from [Stata](#) this page is meant to demonstrate how different Stata operations would be performed in pandas.

If you're new to pandas, you might want to first read through [10 Minutes to pandas](#) to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows. This means that we can refer to the libraries as `pd` and `np`, respectively, for the rest of the document.

```
In [1]: import pandas as pd
```

```
In [2]: import numpy as np
```

---

**Note:** Throughout this tutorial, the pandas `DataFrame` will be displayed by calling `df.head()`, which displays the first N (default 5) rows of the `DataFrame`. This is often used in interactive work (e.g. [Jupyter notebook](#) or terminal) – the equivalent in Stata would be:

```
list in 1/5
```

---

### 33.1 Data Structures

#### 33.1.1 General Terminology Translation

pandas	Stata
<code>DataFrame</code>	data set
column	variable
row	observation
<code>groupby</code>	<code>bysort</code>
NaN	.

#### 33.1.2 `DataFrame` / `Series`

A `DataFrame` in pandas is analogous to a Stata data set – a two-dimensional data source with labeled columns that can be of different types. As will be shown in this document, almost any operation that can be applied to a data set in Stata can also be accomplished in pandas.

A `Series` is the data structure that represents one column of a `DataFrame`. Stata doesn't have a separate data structure for a single column, but in general, working with a `Series` is analogous to referencing a column of a data set in Stata.

### 33.1.3 Index

Every `DataFrame` and `Series` has an `Index` – labels on the *rows* of the data. Stata does not have an exactly analogous concept. In Stata, a data set's rows are essentially unlabeled, other than an implicit integer index that can be accessed with `_n`.

In pandas, if no index is specified, an integer index is also used by default (first row = 0, second row = 1, and so on). While using a labeled `Index` or `MultiIndex` can enable sophisticated analyses and is ultimately an important part of pandas to understand, for this comparison we will essentially ignore the `Index` and just treat the `DataFrame` as a collection of columns. Please see the [indexing documentation](#) for much more on how to use an `Index` effectively.

## 33.2 Data Input / Output

### 33.2.1 Constructing a DataFrame from Values

A Stata data set can be built from specified values by placing the data after an `input` statement and specifying the column names.

```
input x y
1 2
3 4
5 6
end
```

A pandas `DataFrame` can be constructed in many different ways, but for a small number of values, it is often convenient to specify it as a Python dictionary, where the keys are the column names and the values are the data.

```
In [3]: df = pd.DataFrame({
...:     'x': [1, 3, 5],
...:     'y': [2, 4, 6]})
...:

In [4]: df
Out[4]:
   x  y
0  1  2
1  3  4
2  5  6
```

### 33.2.2 Reading External Data

Like Stata, pandas provides utilities for reading in data from many formats. The `tips` data set, found within the pandas tests (`csv`) will be used in many of the following examples.

Stata provides `import delimited` to read `csv` data into a data set in memory. If the `tips.csv` file is in the current working directory, we can import it as follows.

```
import delimited tips.csv
```

The pandas method is `read_csv()`, which works similarly. Additionally, it will automatically download the data set if presented with a url.

```
In [5]: url = 'https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/data/tips.
↳ csv'

In [6]: tips = pd.read_csv(url)

In [7]: tips.head()
Out[7]:
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

Like `import delimited`, `read_csv()` can take a number of parameters to specify how the data should be parsed. For example, if the data were instead tab delimited, did not have column names, and existed in the current working directory, the pandas command would be:

```
tips = pd.read_csv('tips.csv', sep='\t', header=None)

# alternatively, read_table is an alias to read_csv with tab delimiter
tips = pd.read_table('tips.csv', header=None)
```

Pandas can also read Stata data sets in `.dta` format with the `read_stata()` function.

```
df = pd.read_stata('data.dta')
```

In addition to text/csv and Stata files, pandas supports a variety of other data formats such as Excel, SAS, HDF5, Parquet, and SQL databases. These are all read via a `pd.read_*` function. See the [IO documentation](#) for more details.

### 33.2.3 Exporting Data

The inverse of `import delimited` in Stata is `export delimited`

```
export delimited tips2.csv
```

Similarly in pandas, the opposite of `read_csv` is `DataFrame.to_csv()`.

```
tips.to_csv('tips2.csv')
```

Pandas can also export to Stata file format with the `DataFrame.to_stata()` method.

```
tips.to_stata('tips2.dta')
```

## 33.3 Data Operations

### 33.3.1 Operations on Columns

In Stata, arbitrary math expressions can be used with the `generate` and `replace` commands on new or existing columns. The `drop` command drops the column from the data set.

```

replace total_bill = total_bill - 2
generate new_bill = total_bill / 2
drop new_bill

```

pandas provides similar vectorized operations by specifying the individual Series in the DataFrame. New columns can be assigned in the same way. The `DataFrame.drop()` method drops a column from the DataFrame.

```

In [8]: tips['total_bill'] = tips['total_bill'] - 2

In [9]: tips['new_bill'] = tips['total_bill'] / 2

In [10]: tips.head()
Out[10]:
   total_bill  tip  sex smoker  day  time  size  new_bill
0      14.99  1.01 Female    No  Sun  Dinner     2     7.495
1       8.34  1.66   Male    No  Sun  Dinner     3     4.170
2      19.01  3.50   Male    No  Sun  Dinner     3     9.505
3      21.68  3.31   Male    No  Sun  Dinner     2    10.840
4      22.59  3.61 Female    No  Sun  Dinner     4    11.295

In [11]: tips = tips.drop('new_bill', axis=1)

```

### 33.3.2 Filtering

Filtering in Stata is done with an `if` clause on one or more columns.

```

list if total_bill > 10

```

DataFrames can be filtered in multiple ways; the most intuitive of which is using *boolean indexing*.

```

In [12]: tips[tips['total_bill'] > 10].head()
Out[12]:
   total_bill  tip  sex smoker  day  time  size
0      14.99  1.01 Female    No  Sun  Dinner     2
2      19.01  3.50   Male    No  Sun  Dinner     3
3      21.68  3.31   Male    No  Sun  Dinner     2
4      22.59  3.61 Female    No  Sun  Dinner     4
5      23.29  4.71   Male    No  Sun  Dinner     4

```

### 33.3.3 If/Then Logic

In Stata, an `if` clause can also be used to create new columns.

```

generate bucket = "low" if total_bill < 10
replace bucket = "high" if total_bill >= 10

```

The same operation in pandas can be accomplished using the `where` method from numpy.

```

In [13]: tips['bucket'] = np.where(tips['total_bill'] < 10, 'low', 'high')

In [14]: tips.head()
Out[14]:
   total_bill  tip  sex smoker  day  time  size bucket
0      14.99  1.01 Female    No  Sun  Dinner     2   high

```

(continues on next page)

(continued from previous page)

1	8.34	1.66	Male	No	Sun	Dinner	3	low
2	19.01	3.50	Male	No	Sun	Dinner	3	high
3	21.68	3.31	Male	No	Sun	Dinner	2	high
4	22.59	3.61	Female	No	Sun	Dinner	4	high

### 33.3.4 Date Functionality

Stata provides a variety of functions to do operations on date/datetime columns.

```

generate date1 = mdy(1, 15, 2013)
generate date2 = date("Feb152015", "MDY")

generate date1_year = year(date1)
generate date2_month = month(date2)

* shift date to beginning of next month
generate date1_next = mdy(month(date1) + 1, 1, year(date1)) if month(date1) != 12
replace date1_next = mdy(1, 1, year(date1) + 1) if month(date1) == 12
generate months_between = mofd(date2) - mofd(date1)

list date1 date2 date1_year date2_month date1_next months_between

```

The equivalent pandas operations are shown below. In addition to these functions, pandas supports other Time Series features not available in Stata (such as time zone handling and custom offsets) – see the [timeseries documentation](#) for more details.

```

In [15]: tips['date1'] = pd.Timestamp('2013-01-15')

In [16]: tips['date2'] = pd.Timestamp('2015-02-15')

In [17]: tips['date1_year'] = tips['date1'].dt.year

In [18]: tips['date2_month'] = tips['date2'].dt.month

In [19]: tips['date1_next'] = tips['date1'] + pd.offsets.MonthBegin()

In [20]: tips['months_between'] = (tips['date2'].dt.to_period('M') -
....:                             tips['date1'].dt.to_period('M'))
....:

In [21]: tips[['date1', 'date2', 'date1_year', 'date2_month',
....:          'date1_next', 'months_between']].head()
....:

Out[21]:
   date1      date2  date1_year  date2_month  date1_next  months_between
0 2013-01-15 2015-02-15        2013          2 2013-02-01              25
1 2013-01-15 2015-02-15        2013          2 2013-02-01              25
2 2013-01-15 2015-02-15        2013          2 2013-02-01              25
3 2013-01-15 2015-02-15        2013          2 2013-02-01              25
4 2013-01-15 2015-02-15        2013          2 2013-02-01              25

```

### 33.3.5 Selection of Columns

Stata provides keywords to select, drop, and rename columns.

```

keep sex total_bill tip

drop sex

rename total_bill total_bill_2

```

The same operations are expressed in pandas below. Note that in contrast to Stata, these operations do not happen in place. To make these changes persist, assign the operation back to a variable.

```

# keep
In [22]: tips[['sex', 'total_bill', 'tip']].head()
Out[22]:
   sex  total_bill  tip
0  Female      14.99  1.01
1   Male       8.34  1.66
2   Male      19.01  3.50
3   Male      21.68  3.31
4  Female      22.59  3.61

# drop
In [23]: tips.drop('sex', axis=1).head()
Out[23]:
   total_bill  tip  smoker  day  time  size
0      14.99  1.01     No  Sun  Dinner    2
1       8.34  1.66     No  Sun  Dinner    3
2      19.01  3.50     No  Sun  Dinner    3
3      21.68  3.31     No  Sun  Dinner    2
4      22.59  3.61     No  Sun  Dinner    4

# rename
In [24]: tips.rename(columns={'total_bill': 'total_bill_2'}).head()
Out[24]:
   total_bill_2  tip  sex  smoker  day  time  size
0      14.99  1.01  Female     No  Sun  Dinner    2
1       8.34  1.66   Male     No  Sun  Dinner    3
2      19.01  3.50   Male     No  Sun  Dinner    3
3      21.68  3.31   Male     No  Sun  Dinner    2
4      22.59  3.61  Female     No  Sun  Dinner    4

```

### 33.3.6 Sorting by Values

Sorting in Stata is accomplished via `sort`

```
sort sex total_bill
```

pandas objects have a `DataFrame.sort_values()` method, which takes a list of columns to sort by.

```

In [25]: tips = tips.sort_values(['sex', 'total_bill'])
In [26]: tips.head()
Out[26]:
   total_bill  tip  sex  smoker  day  time  size
67       1.07  1.00  Female     Yes  Sat  Dinner    1

```

(continues on next page)

(continued from previous page)

92	3.75	1.00	Female	Yes	Fri	Dinner	2
111	5.25	1.00	Female	No	Sat	Dinner	1
145	6.35	1.50	Female	No	Thur	Lunch	2
135	6.51	1.25	Female	No	Thur	Lunch	2

```
generate strlen_time = strlen(time)
generate ustrlen_time = ustrlen(time)
```

---

```
In [27]: tips['time'].str.len().head()
Out[27]:
67      6
92      6
111     6
145     5
135     5
Name: time, dtype: int64

In [28]: tips['time'].str.rstrip().str.len().head()
Out[28]:
67      6
92      6
111     6
145     5
135     5
Name: time, dtype: int64
```

```
generate str_position = strpos(sex, "ale")
```

---

```
In [29]: tips['sex'].str.find("ale").head()
Out[29]:
67      3
```

(continues on next page)

(continued from previous page)

```
92      3
111     3
145     3
135     3
Name: sex, dtype: int64
```

### 33.4.3 Extracting Substring by Position

Stata extracts a substring from a string based on its position with the `substr()` function.

```
generate short_sex = substr(sex, 1, 1)
```

With pandas you can use `[]` notation to extract a substring from a string by position locations. Keep in mind that Python indexes are zero-based.

```
In [30]: tips['sex'].str[0:1].head()
Out[30]:
67      F
92      F
111     F
145     F
135     F
Name: sex, dtype: object
```

### 33.4.4 Extracting nth Word

The Stata `word()` function returns the *nth* word from a string. The first argument is the string you want to parse and the second argument specifies which word you want to extract.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate first_name = word(name, 1)
generate last_name = word(name, -1)
```

Python extracts a substring from a string based on its text by using regular expressions. There are much more powerful approaches, but this just shows a simple approach.

```
In [31]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [32]: firstlast['First_Name'] = firstlast['string'].str.split(" ", expand=True)[0]

In [33]: firstlast['Last_Name'] = firstlast['string'].str.rsplit(" ", expand=True)[0]

In [34]: firstlast
Out[34]:
      string First_Name Last_Name
0  John Smith      John      John
1  Jane Cook      Jane      Jane
```



### 33.4.5 Changing Case

The Stata `strupper()`, `strlower()`, `strproper()`, `ustrupper()`, `ustrlower()`, and `ustrtitle()` functions change the case of ASCII and Unicode strings, respectively.

```
clear
input str20 string
"John Smith"
"Jane Cook"
end

generate upper = strupper(string)
generate lower = strlower(string)
generate title = strproper(string)
list
```

The equivalent Python functions are `upper`, `lower`, and `title`.

```
In [35]: firstlast = pd.DataFrame({'string': ['John Smith', 'Jane Cook']})

In [36]: firstlast['upper'] = firstlast['string'].str.upper()

In [37]: firstlast['lower'] = firstlast['string'].str.lower()

In [38]: firstlast['title'] = firstlast['string'].str.title()

In [39]: firstlast
Out[39]:
```

	string	upper	lower	title
0	John Smith	JOHN SMITH	john smith	John Smith
1	Jane Cook	JANE COOK	jane cook	Jane Cook

## 33.5 Merging

The following tables will be used in the merge examples

```
In [40]: df1 = pd.DataFrame({'key': ['A', 'B', 'C', 'D'],
.....:                      'value': np.random.randn(4)})
.....:

In [41]: df1
Out[41]:
```

	key	value
0	A	0.885906
1	B	0.794848
2	C	-0.943848
3	D	0.328609

```
In [42]: df2 = pd.DataFrame({'key': ['B', 'D', 'D', 'E'],
.....:                      'value': np.random.randn(4)})
.....:

In [43]: df2
Out[43]:
```

	key	value
--	-----	-------

(continues on next page)

(continued from previous page)

0	B	-1.634931
1	D	2.197567
2	D	0.054695
3	E	0.283297

In Stata, to perform a merge, one data set must be in memory and the other must be referenced as a file name on disk. In contrast, Python must have both `DataFrames` already in memory.

By default, Stata performs an outer join, where all observations from both data sets are left in memory after the merge. One can keep only observations from the initial data set, the merged data set, or the intersection of the two by using the values created in the `_merge` variable.

```
* First create df2 and save to disk
clear
input str1 key
B
D
D
E
end
generate value = rnormal()
save df2.dta

* Now create df1 in memory
clear
input str1 key
A
B
C
D
end
generate value = rnormal()

preserve

* Left join
merge 1:n key using df2.dta
keep if _merge == 1

* Right join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 2

* Inner join
restore, preserve
merge 1:n key using df2.dta
keep if _merge == 3

* Outer join
restore
merge 1:n key using df2.dta
```

pandas `DataFrames` have a `DataFrame.merge()` method, which provides similar functionality. Note that different join types are accomplished via the `how` keyword.

```
In [44]: inner_join = df1.merge(df2, on=['key'], how='inner')
```

```
In [45]: inner_join
```

```
Out[45]:
```

	key	value_x	value_y
0	B	0.794848	-1.634931
1	D	0.328609	2.197567
2	D	0.328609	0.054695

```
In [46]: left_join = df1.merge(df2, on=['key'], how='left')
```

```
In [47]: left_join
```

```
Out[47]:
```

	key	value_x	value_y
0	A	0.885906	NaN
1	B	0.794848	-1.634931
2	C	-0.943848	NaN
3	D	0.328609	2.197567
4	D	0.328609	0.054695

```
In [48]: right_join = df1.merge(df2, on=['key'], how='right')
```

```
In [49]: right_join
```

```
Out[49]:
```

	key	value_x	value_y
0	B	0.794848	-1.634931
1	D	0.328609	2.197567
2	D	0.328609	0.054695
3	E	NaN	0.283297

```
In [50]: outer_join = df1.merge(df2, on=['key'], how='outer')
```

```
In [51]: outer_join
```

```
Out[51]:
```

	key	value_x	value_y
0	A	0.885906	NaN
1	B	0.794848	-1.634931
2	C	-0.943848	NaN
3	D	0.328609	2.197567
4	D	0.328609	0.054695
5	E	NaN	0.283297

## 33.6 Missing Data

Like Stata, pandas has a representation for missing data – the special float value NaN (not a number). Many of the semantics are the same; for example missing data propagates through numeric operations, and is ignored by default for aggregations.

```
In [52]: outer_join
```

```
Out[52]:
```

	key	value_x	value_y
0	A	0.885906	NaN
1	B	0.794848	-1.634931
2	C	-0.943848	NaN
3	D	0.328609	2.197567

(continues on next page)

(continued from previous page)

4	D	0.328609	0.054695
5	E	NaN	0.283297

```
In [53]: outer_join['value_x'] + outer_join['value_y']
```

```
0      NaN
1   -0.840083
2      NaN
3    2.526176
4    0.383304
5      NaN
dtype: float64
```

```
In [54]: outer_join['value_x'].sum()
```

---

→ 1.3941243310085349

One difference is that missing data cannot be compared to its sentinel value. For example, in Stata you could do this to filter missing values.

```
* Keep missing values
list if value_x == .
* Keep non-missing values
list if value_x != .
```

This doesn't work in pandas. Instead, the `pd.isna()` or `pd.notna()` functions should be used for comparisons.

```
In [55]: outer_join[pd.isna(outer_join['value_x'])]
```

Out [55]:

	key	value_x	value_y
5	E	NaN	0.283297

```
In [56]: outer_join(pd.notna(outer_join['value_x']))
```

```
Out[56]:
```

	key	value_x	value_y
0	A	0.885906	NaN
1	B	0.794848	-1.634931
2	C	-0.943848	NaN
3	D	0.328609	2.197567
4	D	0.328609	0.054695

Pandas also provides a variety of methods to work with missing data – some of which would be challenging to express in Stata. For example, there are methods to drop all rows with any missing values, replacing missing values with a specified value, like the mean, or forward filling from previous rows. See the [missing data documentation](#) for more.

```
# Drop rows with any missing value
```

```
In [57]: outer_join.dropna()
```

Out [57]:

	key	value_x	value_y
1	B	0.794848	-1.634931
3	D	0.328609	2.197567
4	D	0.328609	0.054695

```
# Fill forwards
```

```
In [58]: outer_join.fillna(method='ffill')
```

(continues on next page)

(continued from previous page)

```

////////////////////////////////////
↪
   key  value_x  value_y
0    A  0.885906      NaN
1    B  0.794848 -1.634931
2    C -0.943848 -1.634931
3    D  0.328609  2.197567
4    D  0.328609  0.054695
5    E  0.328609  0.283297

# Impute missing values with the mean
In [59]: outer_join['value_x'].fillna(outer_join['value_x'].mean())
////////////////////////////////////
↪
0    0.885906
1    0.794848
2   -0.943848
3    0.328609
4    0.328609
5    0.278825
Name: value_x, dtype: float64

```

## 33.7 GroupBy

### 33.7.1 Aggregation

Stata's `collapse` can be used to group by one or more key variables and compute aggregations on numeric columns.

```
collapse (sum) total_bill tip, by(sex smoker)
```

pandas provides a flexible `groupby` mechanism that allows similar aggregations. See the [groupby documentation](#) for more details and examples.

```
In [60]: tips_summed = tips.groupby(['sex', 'smoker'])['total_bill', 'tip'].sum()
```

```
In [61]: tips_summed.head()
```

```
Out [61]:
```

		total_bill	tip
sex	smoker		
Female	No	869.68	149.77
	Yes	527.27	96.74
Male	No	1725.75	302.00
	Yes	1217.07	183.07

### 33.7.2 Transformation

In Stata, if the group aggregations need to be used with the original data set, one would usually use `bysort` with `egen()`. For example, to subtract the mean for each observation by smoker group.

```
bysort sex smoker: egen group_bill = mean(total_bill)
generate adj_total_bill = total_bill - group_bill
```

pandas `groupby` provides a `transform` mechanism that allows these type of operations to be succinctly expressed in one operation.

```
In [62]: gb = tips.groupby('smoker')['total_bill']

In [63]: tips['adj_total_bill'] = tips['total_bill'] - gb.transform('mean')

In [64]: tips.head()
Out[64]:
```

	total_bill	tip	sex	smoker	day	time	size	adj_total_bill
67	1.07	1.00	Female	Yes	Sat	Dinner	1	-17.686344
92	3.75	1.00	Female	Yes	Fri	Dinner	2	-15.006344
111	5.25	1.00	Female	No	Sat	Dinner	1	-11.938278
145	6.35	1.50	Female	No	Thur	Lunch	2	-10.838278
135	6.51	1.25	Female	No	Thur	Lunch	2	-10.678278

### 33.7.3 By Group Processing

In addition to aggregation, pandas `groupby` can be used to replicate most other `bysort` processing from Stata. For example, the following example lists the first observation in the current sort order by sex/smoker group.

```
bysort sex smoker: list if _n == 1
```

In pandas this would be written as:

```
In [65]: tips.groupby(['sex', 'smoker']).first()
Out[65]:
```

sex	smoker	total_bill	tip	day	time	size	adj_total_bill
Female	No	5.25	1.00	Sat	Dinner	1	-11.938278
	Yes	1.07	1.00	Sat	Dinner	1	-17.686344
Male	No	5.51	2.00	Thur	Lunch	2	-11.678278
	Yes	5.25	5.15	Sun	Dinner	2	-13.506344

## 33.8 Other Considerations

### 33.8.1 Disk vs Memory

Pandas and Stata both operate exclusively in memory. This means that the size of data able to be loaded in pandas is limited by your machine's memory. If out of core processing is needed, one possibility is the [dask.dataframe](#) library, which provides a subset of pandas functionality for an on-disk `DataFrame`.

## API REFERENCE

This page gives an overview of all public pandas objects, functions and methods. All classes and functions exposed in `pandas.*` namespace are public.

Some subpackages are public which include `pandas.errors`, `pandas.plotting`, and `pandas.testing`. Public functions in `pandas.io` and `pandas.tseries` submodules are mentioned in the documentation. `pandas.api.types` subpackage holds some public functions related to data types in pandas.

**Warning:** The `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are PRIVATE. Stable functionality in such modules is not guaranteed.

### 34.1 Input/Output

#### 34.1.1 Pickling

---

<code>read_pickle(path[, compression])</code>	Load pickled pandas object (or any object) from file.
---	---

---

##### 34.1.1.1 `pandas.read_pickle`

`pandas.read_pickle(path, compression='infer')`  
Load pickled pandas object (or any object) from file.

**Warning:** Loading pickled data received from untrusted sources can be unsafe. See [here](#).

**Parameters** `path` : str

File path where the pickled object will be loaded.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer', then use gzip, bz2, xz or zip if path ends in '.gz', '.bz2', '.xz', or '.zip' respectively, and no decompression otherwise. Set to None for no decompression.

New in version 0.20.0.

**Returns**

**unpickled** [type of object stored in file]

See also:

`DataFrame.to_pickle` Pickle (serialize) DataFrame object to file.

`Series.to_pickle` Pickle (serialize) Series object to file.

`read_hdf` Read HDF5 file into a DataFrame.

`read_sql` Read SQL query or database table into a DataFrame.

`read_parquet` Load a parquet object, returning a DataFrame.

## Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> pd.to_pickle(original_df, "./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

### 34.1.2 Flat File

<code>read_table(filepath_or_buffer[, sep, ...])</code>	Read general delimited file into DataFrame
<code>read_csv(filepath_or_buffer[, sep, ...])</code>	Read CSV (comma-separated) file into DataFrame
<code>read_fwf(filepath_or_buffer[, colspecs, widths])</code>	Read a table of fixed-width formatted lines into DataFrame
<code>read_msgpack(path_or_buf[, encoding, iterator])</code>	Load msgpack pandas object from the specified file path



### 34.1.2.1 pandas.read\_table

```
pandas.read_table(filepath_or_buffer, sep='\t', delimiter=None, header='infer', names=None,
                  index_col=None, usecols=None, squeeze=False, prefix=None, man-
                  gle_dupe_cols=True, dtype=None, engine=None, converters=None,
                  true_values=None, false_values=None, skipinitialspace=False, skiprows=None,
                  nrows=None, na_values=None, keep_default_na=True, na_filter=True,
                  verbose=False, skip_blank_lines=True, parse_dates=False, infer-
                  datetime_format=False, keep_date_col=False, date_parser=None, day-
                  first=False, iterator=False, chunksize=None, compression='infer', thou-
                  sands=None, decimal=b'.', lineterminator=None, quotechar='"', quot-
                  ing=0, escapechar=None, comment=None, encoding=None, dialect=None,
                  tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skip-
                  footer=0, doublequote=True, delim_whitespace=False, low_memory=True,
                  memory_map=False, float_precision=None)
```

Read general delimited file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the [online docs for IO Tools](#).

#### Parameters

**filepath\_or\_buffer** [str, pathlib.Path, py.\_path.local.LocalPath or any \]

**object with a read() method (such as a file handle or StringIO)**

The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be `file://localhost/path/to/table.csv`

**sep** : str, default t (tab-stop)

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`

**delimiter** : str, default None

Alternative argument name for sep.

**delim\_whitespace** : boolean, default False

Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the sep. Equivalent to setting `sep='\s+'`. If this option is set to True, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

**header** : int or list of ints, default 'infer'

Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if

`skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

**names** : array-like, default None

List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

**index\_col** : int or sequence or False, default None

Column to use as the row labels of the `DataFrame`. If a sequence is given, a `MultiIndex` is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to *not* use the first column as the index (row names)

**usecols** : list-like or callable, default None

Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`. Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a `DataFrame` from data with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

**squeeze** : boolean, default False

If the parsed data only contains one column then return a `Series`

**prefix** : str, default None

Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, ...

**mangle\_dupe\_cols** : boolean, default True

Duplicate columns will be specified as 'X', 'X.1', ... 'X.N', rather than 'X'... 'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

**dtype** : Type name or dict of column -> type, default None

Data type for data or columns. E.g. `{ 'a': np.float64, 'b': np.int32 }` Use *str* or *object* together with suitable *na\_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**engine** : { 'c', 'python' }, optional

Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

**converters** : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**true\_values** : list, default None

Values to consider as True

**false\_values** : list, default None

Values to consider as False

**skipinitialspace** : boolean, default False

Skip spaces after delimiter.

**skiprows** : list-like or integer or callable, default None

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

**skipfooter** : int, default 0

Number of lines at bottom of file to skip (Unsupported with engine='c')

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**na\_values** : scalar, str, list-like, or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: `'', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'`.

**keep\_default\_na** : bool, default True

Whether or not to include the default NaN values when parsing the data. Depending on whether *na\_values* is passed in, the behavior is as follows:

- If *keep\_default\_na* is True, and *na\_values* are specified, *na\_values* is appended to the default NaN values used for parsing.
- If *keep\_default\_na* is True, and *na\_values* are not specified, only the default NaN values are used for parsing.
- If *keep\_default\_na* is False, and *na\_values* are specified, only the NaN values specified *na\_values* are used for parsing.
- If *keep\_default\_na* is False, and *na\_values* are not specified, no strings will be parsed as NaN.

Note that if *na\_filter* is passed in as False, the *keep\_default\_na* and *na\_values* parameters will be ignored.

**na\_filter** : boolean, default True

Detect missing value markers (empty strings and the value of *na\_values*). In data without any NAs, passing *na\_filter=False* can improve the performance of reading a large file

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**skip\_blank\_lines** : boolean, default True

If True, skip over blank lines rather than interpreting as NaN values

**parse\_dates** : boolean or list of ints or names or list of lists or dict, default False

- boolean. If True -> try parsing the index.
- list of ints or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. { 'foo' : [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

**infer\_datetime\_format** : boolean, default False

If True and *parse\_dates* is enabled, pandas will attempt to infer the format of the date-time strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

**keep\_date\_col** : boolean, default False

If True and *parse\_dates* specifies combining multiple columns then keep the original columns.

**date\_parser** : function, default None

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date\_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse\_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse\_dates* into a single array and pass that; and 3) call *date\_parser* once for each row using one or more strings (corresponding to the columns defined by *parse\_dates*) as arguments.

**dayfirst** : boolean, default False

DD/MM format dates, international and European format

**iterator** : boolean, default False

Return `TextFileReader` object for iteration or getting chunks with `get_chunk()`.

**chunksize** : int, default None

Return `TextFileReader` object for iteration. See the [IO Tools docs](#) for more information on *iterator* and *chunksize*.

**compression** : { 'infer', 'gzip', 'bz2', 'zip', 'xz', None }, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer' and *filepath\_or\_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

**thousands** : str, default None

Thousands separator

**decimal** : str, default '.'

Character to recognize as decimal point (e.g. use ‘,’ for European data).

**float\_precision** : string, default None

Specifies which converter the C engine should use for floating-point values. The options are *None* for the ordinary converter, *high* for the high-precision converter, and *round\_trip* for the round-trip converter.

**lineterminator** : str (length 1), default None

Character to break file into lines. Only valid with C parser.

**quotechar** : str (length 1), optional

The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

**quoting** : int or csv.QUOTE\_\* instance, default 0

Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

**doublequote** : boolean, default True

When quotechar is specified and quoting is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive quotechar elements *INSIDE* a field as a single quotechar element.

**escapechar** : str (length 1), default None

One-character string used to escape delimiter when quoting is `QUOTE_NONE`.

**comment** : str, default None

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in ‘a,b,c’ being treated as the header.

**encoding** : str, default None

Encoding to use for UTF when reading/writing (ex. ‘utf-8’). [List of Python standard encodings](#)

**dialect** : str or csv.Dialect instance, default None

If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a `ParserWarning` will be issued. See `csv.Dialect` documentation for more details.

**tupleize\_cols** : boolean, default False

Deprecated since version 0.21.0: This argument will be removed and will always convert to `MultiIndex`

Leave a list of tuples on columns as is (default is to convert to a `MultiIndex` on the columns)

**error\_bad\_lines** : boolean, default True

Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no DataFrame will be returned. If False, then these “bad lines” will be dropped from the DataFrame that is returned.

**warn\_bad\_lines** : boolean, default True

If error\_bad\_lines is False, and warn\_bad\_lines is True, a warning for each “bad line” will be output.

**low\_memory** : boolean, default True

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the *dtype* parameter. Note that the entire file is read into a single DataFrame regardless, use the *chunksize* or *iterator* parameter to return the data in chunks. (Only valid with C parser)

**memory\_map** : boolean, default False

If a filepath is provided for *filepath\_or\_buffer*, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

### Returns

**result** [DataFrame or TextParser]

#### 34.1.2.2 pandas.read\_csv

`pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, doublequote=True, delim_whitespace=False, low_memory=True, memory_map=False, float_precision=None)`

Read CSV (comma-separated) file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the [online docs for IO Tools](#).

### Parameters

**filepath\_or\_buffer** [str, pathlib.Path, py.\_path.local.LocalPath or any \]

**object with a read() method (such as a file handle or StringIO)**

The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be `file://localhost/path/to/table.csv`

**sep** : str, default ','

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically

detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`

**delimiter** : str, default `None`

Alternative argument name for `sep`.

**delim\_whitespace** : boolean, default `False`

Specifies whether or not whitespace (e.g. `' '` or `'\t'`) will be used as the `sep`. Equivalent to setting `sep='\s+'`. If this option is set to `True`, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

**header** : int or list of ints, default `'infer'`

Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

**names** : array-like, default `None`

List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

**index\_col** : int or sequence or `False`, default `None`

Column to use as the row labels of the `DataFrame`. If a sequence is given, a `MultiIndex` is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to `_not_` use the first column as the index (row names)

**usecols** : list-like or callable, default `None`

Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in `names` or inferred from the document header row(s). For example, a valid list-like `usecols` parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`. Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a `DataFrame` from data with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to `True`. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

**squeeze** : boolean, default `False`

If the parsed data only contains one column then return a Series

**prefix** : str, default None

Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, ...

**mangle\_dupe\_cols** : boolean, default True

Duplicate columns will be specified as 'X', 'X.1', ... 'X.N', rather than 'X'... 'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

**dtype** : Type name or dict of column -> type, default None

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use *str* or *object* together with suitable *na\_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**engine** : {'c', 'python'}, optional

Parser engine to use. The C engine is faster while the python engine is currently more feature-complete.

**converters** : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**true\_values** : list, default None

Values to consider as True

**false\_values** : list, default None

Values to consider as False

**skipinitialspace** : boolean, default False

Skip spaces after delimiter.

**skiprows** : list-like or integer or callable, default None

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

**skipfooter** : int, default 0

Number of lines at bottom of file to skip (Unsupported with engine='c')

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**na\_values** : scalar, str, list-like, or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: ' ', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

**keep\_default\_na** : bool, default True

Whether or not to include the default NaN values when parsing the data. Depending on whether *na\_values* is passed in, the behavior is as follows:



- If *keep\_default\_na* is True, and *na\_values* are specified, *na\_values* is appended to the default NaN values used for parsing.
- If *keep\_default\_na* is True, and *na\_values* are not specified, only the default NaN values are used for parsing.
- If *keep\_default\_na* is False, and *na\_values* are specified, only the NaN values specified *na\_values* are used for parsing.
- If *keep\_default\_na* is False, and *na\_values* are not specified, no strings will be parsed as NaN.

Note that if *na\_filter* is passed in as False, the *keep\_default\_na* and *na\_values* parameters will be ignored.

**na\_filter** : boolean, default True

Detect missing value markers (empty strings and the value of *na\_values*). In data without any NAs, passing *na\_filter*=False can improve the performance of reading a large file

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**skip\_blank\_lines** : boolean, default True

If True, skip over blank lines rather than interpreting as NaN values

**parse\_dates** : boolean or list of ints or names or list of lists or dict, default False

- boolean. If True -> try parsing the index.
- list of ints or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. { 'foo' : [1, 3] } -> parse columns 1, 3 as date and call result 'foo'

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

**infer\_datetime\_format** : boolean, default False

If True and *parse\_dates* is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

**keep\_date\_col** : boolean, default False

If True and *parse\_dates* specifies combining multiple columns then keep the original columns.

**date\_parser** : function, default None

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date\_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse\_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse\_dates* into a

single array and pass that; and 3) call *date\_parser* once for each row using one or more strings (corresponding to the columns defined by *parse\_dates*) as arguments.

**dayfirst** : boolean, default False

DD/MM format dates, international and European format

**iterator** : boolean, default False

Return TextFileReader object for iteration or getting chunks with `get_chunk()`.

**chunksize** : int, default None

Return TextFileReader object for iteration. See the [IO Tools docs](#) for more information on `iterator` and `chunksize`.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer' and *filepath\_or\_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

**thousands** : str, default None

Thousands separator

**decimal** : str, default '.'

Character to recognize as decimal point (e.g. use ',' for European data).

**float\_precision** : string, default None

Specifies which converter the C engine should use for floating-point values. The options are *None* for the ordinary converter, *high* for the high-precision converter, and *round\_trip* for the round-trip converter.

**lineterminator** : str (length 1), default None

Character to break file into lines. Only valid with C parser.

**quotechar** : str (length 1), optional

The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

**quoting** : int or csv.QUOTE\_\* instance, default 0

Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

**doublequote** : boolean, default True

When `quotechar` is specified and `quoting` is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements **INSIDE** a field as a single `quotechar` element.

**escapechar** : str (length 1), default None

One-character string used to escape delimiter when `quoting` is `QUOTE_NONE`.

**comment** : str, default None

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter *header* but not by *skiprows*. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in 'a,b,c' being treated as the header.

**encoding** : str, default None

Encoding to use for UTF when reading/writing (ex. 'utf-8'). [List of Python standard encodings](#)

**dialect** : str or csv.Dialect instance, default None

If provided, this parameter will override values (default or not) for the following parameters: *delimiter*, *doublequote*, *escapechar*, *skipinitialspace*, *quotechar*, and *quoting*. If it is necessary to override values, a `ParserWarning` will be issued. See `csv.Dialect` documentation for more details.

**tupleize\_cols** : boolean, default False

Deprecated since version 0.21.0: This argument will be removed and will always convert to `MultiIndex`

Leave a list of tuples on columns as is (default is to convert to a `MultiIndex` on the columns)

**error\_bad\_lines** : boolean, default True

Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no `DataFrame` will be returned. If False, then these “bad lines” will be dropped from the `DataFrame` that is returned.

**warn\_bad\_lines** : boolean, default True

If `error_bad_lines` is False, and `warn_bad_lines` is True, a warning for each “bad line” will be output.

**low\_memory** : boolean, default True

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set False, or specify the type with the *dtype* parameter. Note that the entire file is read into a single `DataFrame` regardless, use the *chunksize* or *iterator* parameter to return the data in chunks. (Only valid with C parser)

**memory\_map** : boolean, default False

If a filepath is provided for *filepath\_or\_buffer*, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.

## Returns

**result** [`DataFrame` or `TextParser`]

### 34.1.2.3 pandas.read\_fwf

`pandas.read_fwf(filepath_or_buffer, colspecs='infer', widths=None, **kws)`

Read a table of fixed-width formatted lines into `DataFrame`

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the [online docs for IO Tools](#).

### Parameters

**filepath\_or\_buffer** [str, pathlib.Path, py.\_path.local.LocalPath or any \]

**object with a read() method (such as a file handle or StringIO)**

The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be `file://localhost/path/to/table.csv`.

**colspecs** : list of pairs (int, int) or 'infer'. optional

A list of pairs (tuples) giving the extents of the fixed-width fields of each line as half-open intervals (i.e., [from, to[ ). String value 'infer' can be used to instruct the parser to try detecting the column specifications from the first 100 rows of the data which are not being skipped via skiprows (default='infer').

**widths** : list of ints. optional

A list of field widths which can be used instead of 'colspecs' if the intervals are contiguous.

**delimiter** : str, default ' ' + ' '

Characters to consider as filler characters in the fixed-width file. Can be used to specify the filler character of the fields if it is not spaces (e.g., '~').

**delim\_whitespace** : boolean, default False

Specifies whether or not whitespace (e.g. ' ' or '\t') will be used as the sep. Equivalent to setting `sep='\s+'`. If this option is set to True, nothing should be passed in for the `delimiter` parameter.

New in version 0.18.1: support for the Python parser.

**header** : int or list of ints, default 'infer'

Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. `[0,1,3]`. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

**names** : array-like, default None

List of column names to use. If file contains no header row, then you should explicitly pass `header=None`. Duplicates in this list will cause a `UserWarning` to be issued.

**index\_col** : int or sequence or False, default None

Column to use as the row labels of the DataFrame. If a sequence is given, a `MultiIndex` is used. If you have a malformed file with delimiters at the end of each line, you might consider `index_col=False` to force pandas to `_not_` use the first column as the index (row names)

**usecols** : list-like or callable, default None

Return a subset of the columns. If list-like, all elements must either be positional (i.e. integer indices into the document columns) or strings that correspond to column names provided either by the user in *names* or inferred from the document header row(s). For example, a valid list-like *usecols* parameter would be `[0, 1, 2]` or `['foo', 'bar', 'baz']`. Element order is ignored, so `usecols=[0, 1]` is the same as `[1, 0]`. To instantiate a DataFrame from data with element order preserved use `pd.read_csv(data, usecols=['foo', 'bar'])[['foo', 'bar']]` for columns in `['foo', 'bar']` order or `pd.read_csv(data, usecols=['foo', 'bar'])[['bar', 'foo']]` for `['bar', 'foo']` order.

If callable, the callable function will be evaluated against the column names, returning names where the callable function evaluates to True. An example of a valid callable argument would be `lambda x: x.upper() in ['AAA', 'BBB', 'DDD']`. Using this parameter results in much faster parsing time and lower memory usage.

**squeeze** : boolean, default False

If the parsed data only contains one column then return a Series

**prefix** : str, default None

Prefix to add to column numbers when no header, e.g. 'X' for X0, X1, ...

**mangle\_dupe\_cols** : boolean, default True

Duplicate columns will be specified as 'X', 'X.1', ... 'X.N', rather than 'X'... 'X'. Passing in False will cause data to be overwritten if there are duplicate names in the columns.

**dtype** : Type name or dict of column -> type, default None

Data type for data or columns. E.g. `{ 'a': np.float64, 'b': np.int32 }` Use *str* or *object* together with suitable *na\_values* settings to preserve and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

**converters** : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**true\_values** : list, default None

Values to consider as True

**false\_values** : list, default None

Values to consider as False

**skipinitialspace** : boolean, default False

Skip spaces after delimiter.

**skiprows** : list-like or integer or callable, default None

Line numbers to skip (0-indexed) or number of lines to skip (int) at the start of the file.

If callable, the callable function will be evaluated against the row indices, returning True if the row should be skipped and False otherwise. An example of a valid callable argument would be `lambda x: x in [0, 2]`.

**skipfooter** : int, default 0

Number of lines at bottom of file to skip (Unsupported with engine='c')

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**na\_values** : scalar, str, list-like, or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: ‘’, ‘#N/A’, ‘#N/A N/A’, ‘#NA’, ‘-1.#IND’, ‘-1.#QNAN’, ‘-NaN’, ‘-nan’, ‘1.#IND’, ‘1.#QNAN’, ‘N/A’, ‘NA’, ‘NULL’, ‘NaN’, ‘n/a’, ‘nan’, ‘null’.

**keep\_default\_na** : bool, default True

Whether or not to include the default NaN values when parsing the data. Depending on whether *na\_values* is passed in, the behavior is as follows:

- If *keep\_default\_na* is True, and *na\_values* are specified, *na\_values* is appended to the default NaN values used for parsing.
- If *keep\_default\_na* is True, and *na\_values* are not specified, only the default NaN values are used for parsing.
- If *keep\_default\_na* is False, and *na\_values* are specified, only the NaN values specified *na\_values* are used for parsing.
- If *keep\_default\_na* is False, and *na\_values* are not specified, no strings will be parsed as NaN.

Note that if *na\_filter* is passed in as False, the *keep\_default\_na* and *na\_values* parameters will be ignored.

**na\_filter** : boolean, default True

Detect missing value markers (empty strings and the value of *na\_values*). In data without any NAs, passing *na\_filter*=False can improve the performance of reading a large file

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**skip\_blank\_lines** : boolean, default True

If True, skip over blank lines rather than interpreting as NaN values

**parse\_dates** : boolean or list of ints or names or list of lists or dict, default False

- boolean. If True -> try parsing the index.
- list of ints or names. e.g. If [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column.
- list of lists. e.g. If [[1, 3]] -> combine columns 1 and 3 and parse as a single date column.
- dict, e.g. { ‘foo’ : [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’

If a column or index contains an unparseable date, the entire column or index will be returned unaltered as an object data type. For non-standard datetime parsing, use `pd.to_datetime` after `pd.read_csv`

Note: A fast-path exists for iso8601-formatted dates.

**infer\_datetime\_format** : boolean, default False

If True and *parse\_dates* is enabled, pandas will attempt to infer the format of the datetime strings in the columns, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by 5-10x.

**keep\_date\_col** : boolean, default False

If True and *parse\_dates* specifies combining multiple columns then keep the original columns.

**date\_parser** : function, default None

Function to use for converting a sequence of string columns to an array of datetime instances. The default uses `dateutil.parser.parser` to do the conversion. Pandas will try to call *date\_parser* in three different ways, advancing to the next if an exception occurs: 1) Pass one or more arrays (as defined by *parse\_dates*) as arguments; 2) concatenate (row-wise) the string values from the columns defined by *parse\_dates* into a single array and pass that; and 3) call *date\_parser* once for each row using one or more strings (corresponding to the columns defined by *parse\_dates*) as arguments.

**dayfirst** : boolean, default False

DD/MM format dates, international and European format

**iterator** : boolean, default False

Return TextFileReader object for iteration or getting chunks with `get_chunk()`.

**chunksize** : int, default None

Return TextFileReader object for iteration. See the [IO Tools docs](#) for more information on *iterator* and *chunksize*.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer' and *filepath\_or\_buffer* is path-like, then detect compression from the following extensions: '.gz', '.bz2', '.zip', or '.xz' (otherwise no decompression). If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.18.1: support for 'zip' and 'xz' compression.

**thousands** : str, default None

Thousands separator

**decimal** : str, default '.'

Character to recognize as decimal point (e.g. use ',' for European data).

**float\_precision** : string, default None

Specifies which converter the C engine should use for floating-point values. The options are *None* for the ordinary converter, *high* for the high-precision converter, and *round\_trip* for the round-trip converter.

**lineterminator** : str (length 1), default None

Character to break file into lines. Only valid with C parser.

**quotechar** : str (length 1), optional

The character used to denote the start and end of a quoted item. Quoted items can include the delimiter and it will be ignored.

**quoting** : int or csv.QUOTE\_\* instance, default 0

Control field quoting behavior per `csv.QUOTE_*` constants. Use one of `QUOTE_MINIMAL` (0), `QUOTE_ALL` (1), `QUOTE_NONNUMERIC` (2) or `QUOTE_NONE` (3).

**doublequote** : boolean, default `True`

When `quotechar` is specified and quoting is not `QUOTE_NONE`, indicate whether or not to interpret two consecutive `quotechar` elements `INSIDE` a field as a single `quotechar` element.

**escapechar** : str (length 1), default `None`

One-character string used to escape delimiter when quoting is `QUOTE_NONE`.

**comment** : str, default `None`

Indicates remainder of line should not be parsed. If found at the beginning of a line, the line will be ignored altogether. This parameter must be a single character. Like empty lines (as long as `skip_blank_lines=True`), fully commented lines are ignored by the parameter `header` but not by `skiprows`. For example, if `comment='#'`, parsing `#empty\na,b,c\n1,2,3` with `header=0` will result in `'a,b,c'` being treated as the header.

**encoding** : str, default `None`

Encoding to use for UTF when reading/writing (ex. `'utf-8'`). [List of Python standard encodings](#)

**dialect** : str or `csv.Dialect` instance, default `None`

If provided, this parameter will override values (default or not) for the following parameters: `delimiter`, `doublequote`, `escapechar`, `skipinitialspace`, `quotechar`, and `quoting`. If it is necessary to override values, a `ParserWarning` will be issued. See `csv.Dialect` documentation for more details.

**tupleize\_cols** : boolean, default `False`

Deprecated since version 0.21.0: This argument will be removed and will always convert to `MultiIndex`

Leave a list of tuples on columns as is (default is to convert to a `MultiIndex` on the columns)

**error\_bad\_lines** : boolean, default `True`

Lines with too many fields (e.g. a csv line with too many commas) will by default cause an exception to be raised, and no `DataFrame` will be returned. If `False`, then these “bad lines” will be dropped from the `DataFrame` that is returned.

**warn\_bad\_lines** : boolean, default `True`

If `error_bad_lines` is `False`, and `warn_bad_lines` is `True`, a warning for each “bad line” will be output.

**low\_memory** : boolean, default `True`

Internally process the file in chunks, resulting in lower memory use while parsing, but possibly mixed type inference. To ensure no mixed types either set `False`, or specify the type with the `dtype` parameter. Note that the entire file is read into a single `DataFrame` regardless, use the `chunksize` or `iterator` parameter to return the data in chunks. (Only valid with C parser)

**memory\_map** : boolean, default `False`

If a `filepath` is provided for `filepath_or_buffer`, map the file object directly onto memory and access the data directly from there. Using this option can improve performance because there is no longer any I/O overhead.



**Returns****result** [DataFrame or TextParser]**34.1.2.4 pandas.read\_msgpack**`pandas.read_msgpack(path_or_buf, encoding='utf-8', iterator=False, **kwargs)`

Load msgpack pandas object from the specified file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

**Parameters****path\_or\_buf** [string File path, BytesIO like or string]**encoding:** Encoding for decoding msgpack str type**iterator** : boolean, if True, return an iterator to the unpacker  
(default is False)**Returns****obj** [type of object stored in file]**34.1.3 Clipboard**`read_clipboard([sep])`

Read text from clipboard and pass to read\_table.

**34.1.3.1 pandas.read\_clipboard**`pandas.read_clipboard(sep='\s+', **kwargs)`

Read text from clipboard and pass to read\_table. See read\_table for the full argument list

**Parameters** **sep** : str, default 's+'.

A string or regex delimiter. The default of 's+' denotes one or more whitespace characters.

**Returns****parsed** [DataFrame]**34.1.4 Excel**`read_excel(io[, sheet_name, header, names, ...])`

Read an Excel table into a pandas DataFrame

`ExcelFile.parse(sheet_name, header, names, ...)`

Parse specified sheet(s) into a DataFrame

### 34.1.4.1 pandas.read\_excel

```
pandas.read_excel(io, sheet_name=0, header=0, names=None, index_col=None, usecols=None,
                  squeeze=False, dtype=None, engine=None, converters=None, true_values=None,
                  false_values=None, skiprows=None, nrows=None, na_values=None,
                  parse_dates=False, date_parser=None, thousands=None, comment=None,
                  skipfooter=0, convert_float=True, **kwargs)
```

Read an Excel table into a pandas DataFrame

**Parameters** **io** : string, path object (pathlib.Path or py.\_path.local.LocalPath),

file-like object, pandas ExcelFile, or xlrd workbook. The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be <file://localhost/path/to/workbook.xlsx>

**sheet\_name** : string, int, mixed list of strings/int, or None, default 0

Strings are used for sheet names, Integers are used in zero-indexed sheet positions.

Lists of strings/integers are used to request multiple sheets.

Specify None to get all sheets.

str/int -> DataFrame is returned. list/None -> Dict of DataFrames is returned, with keys representing sheets.

Available Cases

- Defaults to 0 -> 1st sheet as a DataFrame
- 1 -> 2nd sheet as a DataFrame
- "Sheet1" -> 1st sheet as a DataFrame
- [0,1,"Sheet5"] -> 1st, 2nd & 5th sheet as a dictionary of DataFrames
- None -> All sheets as a dictionary of DataFrames

**sheetname** : string, int, mixed list of strings/int, or None, default 0

Deprecated since version 0.21.0: Use *sheet\_name* instead

**header** : int, list of ints, default 0

Row (0-indexed) to use for the column labels of the parsed DataFrame. If a list of integers is passed those row positions will be combined into a `MultiIndex`. Use None if there is no header.

**names** : array-like, default None

List of column names to use. If file contains no header row, then you should explicitly pass `header=None`

**index\_col** : int, list of ints, default None

Column (0-indexed) to use as the row labels of the DataFrame. Pass None if there is no such column. If a list is passed, those columns will be combined into a `MultiIndex`. If a subset of data is selected with `usecols`, `index_col` is based on the subset.

**parse\_cols** : int or list, default None

Deprecated since version 0.21.0: Pass in *usecols* instead.

**usecols** : int or list, default None

- If None then parse all columns,

- If int then indicates last column to be parsed
- If list of ints then indicates list of column numbers to be parsed
- If string then indicates comma separated list of Excel column letters and column ranges (e.g. "A:E" or "A,C,E:F"). Ranges are inclusive of both sides.

**squeeze** : boolean, default False

If the parsed data only contains one column then return a Series

**dtype** : Type name or dict of column -> type, default None

Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32} Use *object* to preserve data as stored in Excel and not interpret dtype. If converters are specified, they will be applied INSTEAD of dtype conversion.

New in version 0.20.0.

**engine**: string, default None

If io is not a buffer or path, this must be set to identify io. Acceptable values are None or xlrd

**converters** : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the Excel cell content, and return the transformed content.

**true\_values** : list, default None

Values to consider as True

New in version 0.19.0.

**false\_values** : list, default None

Values to consider as False

New in version 0.19.0.

**skiprows** : list-like

Rows to skip at the beginning (0-indexed)

**nrows** : int, default None

Number of rows to parse

New in version 0.23.0.

**na\_values** : scalar, str, list-like, or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA values. By default the following values are interpreted as NaN: '', '#N/A', '#N/A N/A', '#NA', '-1.#IND', '-1.#QNAN', '-NaN', '-nan', '1.#IND', '1.#QNAN', 'N/A', 'NA', 'NULL', 'NaN', 'n/a', 'nan', 'null'.

**keep\_default\_na** : bool, default True

If na\_values are specified and keep\_default\_na is False the default NaN values are overridden, otherwise they're appended to.

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**thousands** : str, default None

Thousands separator for parsing string columns to numeric. Note that this parameter is only necessary for columns stored as TEXT in Excel, any numeric columns will automatically be parsed, regardless of display format.

**comment** : str, default None

Comments out remainder of line. Pass a character or characters to this argument to indicate comments in the input file. Any data between the comment string and the end of the current line is ignored.

**skip\_footer** : int, default 0

Deprecated since version 0.23.0: Pass in *skipfooter* instead.

**skipfooter** : int, default 0

Rows at the end to skip (0-indexed)

**convert\_float** : boolean, default True

convert integral floats to int (i.e., 1.0 → 1). If False, all numeric data will be read in as floats: Excel stores all numbers as floats internally

**Returns** **parsed** : DataFrame or Dict of DataFrames

DataFrame from the passed in Excel file. See notes in *sheet\_name* argument for more information on when a Dict of Dataframes is returned.

## Examples

An example DataFrame written to a local file

```
>>> df_out = pd.DataFrame([('string1', 1),
...                        ('string2', 2),
...                        ('string3', 3)],
...                        columns=['Name', 'Value'])
>>> df_out
   Name  Value
0 string1     1
1 string2     2
2 string3     3
>>> df_out.to_excel('tmp.xlsx')
```

The file can be read using the file name as string or an open file object:

```
>>> pd.read_excel('tmp.xlsx')
   Name  Value
0 string1     1
1 string2     2
2 string3     3
```

```
>>> pd.read_excel(open('tmp.xlsx', 'rb'))
   Name  Value
0 string1     1
1 string2     2
2 string3     3
```

Index and header can be specified via the *index\_col* and *header* arguments

```
>>> pd.read_excel('tmp.xlsx', index_col=None, header=None)
   0      1      2
0 NaN    Name  Value
1 0.0  string1     1
2 1.0  string2     2
3 2.0  string3     3
```

Column types are inferred but can be explicitly specified

```
>>> pd.read_excel('tmp.xlsx', dtype={'Name':str, 'Value':float})
   Name  Value
0 string1   1.0
1 string2   2.0
2 string3   3.0
```

True, False, and NA values, and thousands separators have defaults, but can be explicitly specified, too. Supply the values you would like as strings or lists of strings!

```
>>> pd.read_excel('tmp.xlsx',
...               na_values=['string1', 'string2'])
   Name  Value
0    NaN     1
1    NaN     2
2 string3     3
```

Comment lines in the excel input file can be skipped using the *comment* kwarg

```
>>> df = pd.DataFrame({'a': ['1', '#2'], 'b': ['2', '3']})
>>> df.to_excel('tmp.xlsx', index=False)
>>> pd.read_excel('tmp.xlsx')
   a  b
0  1  2
1 #2  3
```

```
>>> pd.read_excel('tmp.xlsx', comment='#')
   a  b
0  1  2
```

#### 34.1.4.2 pandas.ExcelFile.parse

`ExcelFile.parse(sheet_name=0, header=0, names=None, index_col=None, usecols=None, squeeze=False, converters=None, true_values=None, false_values=None, skiprows=None, nrows=None, na_values=None, parse_dates=False, date_parser=None, thousands=None, comment=None, skipfooter=0, convert_float=True, **kws)`

Parse specified sheet(s) into a DataFrame

Equivalent to `read_excel(ExcelFile, ...)` See the `read_excel` docstring for more info on accepted parameters

#### 34.1.5 JSON

---

<code>read_json([path_or_buf, orient, typ, dtype, ...])</code>	Convert a JSON string to pandas object
--	--

---

### 34.1.5.1 pandas.read\_json

`pandas.read_json` (*path\_or\_buf=None, orient=None, typ='frame', dtype=True, convert\_axes=True, convert\_dates=True, keep\_default\_dates=True, numpy=False, precise\_float=False, date\_unit=None, encoding=None, lines=False, chunksize=None, compression='infer'*)

Convert a JSON string to pandas object

**Parameters** `path_or_buf` : a valid JSON string or file-like, default: None

The string could be a URL. Valid URL schemes include http, ftp, s3, and file. For file URLs, a host is expected. For instance, a local file could be `file://localhost/path/to/table.json`

**orient** : string,

Indication of expected JSON string format. Compatible JSON strings can be produced by `to_json()` with a corresponding `orient` value. The set of possible orients is:

- `'split'` : dict like {index -> [index], columns -> [columns], data -> [values]}
- `'records'` : list like [{column -> value}, ... , {column -> value}]
- `'index'` : dict like {index -> {column -> value}}
- `'columns'` : dict like {column -> {index -> value}}
- `'values'` : just the values array

The allowed and default values depend on the value of the `typ` parameter.

- when `typ == 'series'`,
  - allowed orients are `{'split', 'records', 'index'}`
  - default is `'index'`
  - The Series index must be unique for orient `'index'`.
- when `typ == 'frame'`,
  - allowed orients are `{'split', 'records', 'index', 'columns', 'values', 'table'}`
  - default is `'columns'`
  - The DataFrame index must be unique for orients `'index'` and `'columns'`.
  - The DataFrame columns must be unique for orients `'index'`, `'columns'`, and `'records'`.

New in version 0.23.0: `'table'` as an allowed value for the `orient` argument

**typ** [type of object to recover (series or frame), default `'frame'`]

**dtype** : boolean or dict, default `True`

If `True`, infer dtypes, if a dict of column to dtype, then use those, if `False`, then don't infer dtypes at all, applies only to the data.

**convert\_axes** : boolean, default `True`

Try to convert the axes to the proper dtypes.

**convert\_dates** : boolean, default True

List of columns to parse for dates; If True, then try to parse datelike columns default is True; a column label is datelike if

- it ends with '`_at`',
- it ends with '`_time`',
- it begins with '`timestamp`',
- it is '`modified`', or
- it is '`date`'

**keep\_default\_dates** : boolean, default True

If parsing dates, then parse the default datelike columns

**numpy** : boolean, default False

Direct decoding to numpy arrays. Supports numeric data only, but non-numeric column and index labels are supported. Note also that the JSON ordering **MUST** be the same for each term if `numpy=True`.

**precise\_float** : boolean, default False

Set to enable usage of higher precision (`strtod`) function when decoding string to double values. Default (False) is to use fast but less precise builtin functionality

**date\_unit** : string, default None

The timestamp unit to detect if converting dates. The default behaviour is to try and detect the correct precision, but if this is not desired then pass one of '`s`', '`ms`', '`us`' or '`ns`' to force parsing only seconds, milliseconds, microseconds or nanoseconds respectively.

**lines** : boolean, default False

Read the file as a json object per line.

New in version 0.19.0.

**encoding** : str, default is '`utf-8`'

The encoding to use to decode py3 bytes.

New in version 0.19.0.

**chunksize**: integer, default None

Return `JsonReader` object for iteration. See the [line-delimited json docs](#) for more information on `chunksize`. This can only be passed if `lines=True`. If this is None, the file will be read into memory all at once.

New in version 0.21.0.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

For on-the-fly decompression of on-disk data. If 'infer', then use `gzip`, `bz2`, `zip` or `xz` if `path_or_buf` is a string ending in '`.gz`', '`.bz2`', '`.zip`', or '`.xz`', respectively, and no decompression otherwise. If using 'zip', the ZIP file must contain only one data file to be read in. Set to None for no decompression.

New in version 0.21.0.

## Returns

**result** [Series or DataFrame, depending on the value of *typ*.]

**See also:**`DataFrame.to_json`**Notes**

Specific to `orient='table'`, if a `DataFrame` with a literal `Index` name of `index` gets written with `to_json()`, the subsequent read operation will incorrectly set the `Index` name to `None`. This is because `index` is also used by `DataFrame.to_json()` to denote a missing `Index` name, and the subsequent `read_json()` operation cannot distinguish between the two. The same limitation is encountered with a `MultiIndex` and any names beginning with `'level_'`.

**Examples**

```
>>> df = pd.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                    index=[ 'row 1', 'row 2'],
...                    columns=[ 'col 1', 'col 2'])
```

Encoding/decoding a Dataframe using `'split'` formatted JSON:

```
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]}]}'
>>> pd.read_json(_, orient='split')
   col 1 col 2
row 1    a    b
row 2    c    d
```

Encoding/decoding a Dataframe using `'index'` formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
>>> pd.read_json(_, orient='index')
   col 1 col 2
row 1    a    b
row 2    c    d
```

Encoding/decoding a Dataframe using `'records'` formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"}, {"col 1":"c","col 2":"d"}]'
>>> pd.read_json(_, orient='records')
   col 1 col 2
0      a    b
1      c    d
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
```

(continues on next page)



(continued from previous page)

```

        "primaryKey": "index",
        "pandas_version": "0.20.0"},
    "data": [{ "index": "row 1", "col 1": "a", "col 2": "b"},
              { "index": "row 2", "col 1": "c", "col 2": "d"}]}'

```

<code>json_normalize(data[, record_path, meta, ...])</code>	“Normalize” semi-structured JSON data into a flat table
<code>build_table_schema(data[, index, ...])</code>	Create a Table schema from data.

### 34.1.5.2 pandas.io.json.json\_normalize

`pandas.io.json.json_normalize` (*data*, *record\_path*=None, *meta*=None, *meta\_prefix*=None, *record\_prefix*=None, *errors*='raise', *sep*='.')

“Normalize” semi-structured JSON data into a flat table

**Parameters** *data* : dict or list of dicts

Unserialized JSON objects

**record\_path** : string or list of strings, default None

Path in each object to list of records. If not passed, data will be assumed to be an array of records

**meta** : list of paths (string or list of strings), default None

Fields to use as metadata for each record in resulting table

**record\_prefix** : string, default None

If True, prefix records with dotted (?) path, e.g. foo.bar.field if path to records is ['foo', 'bar']

**meta\_prefix** [string, default None]

**errors** : {'raise', 'ignore'}, default 'raise'

- 'ignore' : will ignore KeyError if keys listed in meta are not always present
- 'raise' : will raise KeyError if keys listed in meta are not always present

New in version 0.20.0.

**sep** : string, default '.'

Nested records will generate names separated by sep, e.g., for sep='.', { 'foo' : { 'bar' : 0 } } -> foo.bar

New in version 0.20.0.

**Returns**

**frame** [DataFrame]

### Examples

```
>>> from pandas.io.json import json_normalize
>>> data = [{'id': 1, 'name': {'first': 'Coleen', 'last': 'Volk'}},
...         {'name': {'given': 'Mose', 'family': 'Regner'}},
...         {'id': 2, 'name': 'Faye Raker'}]
>>> json_normalize(data)
   id      name name.family name.first name.given name.last
0  1.0      NaN      NaN    Coleen      NaN    Volk
1  NaN      NaN    Regner      NaN    Mose      NaN
2  2.0  Faye Raker      NaN      NaN      NaN      NaN
```

```
>>> data = [{'state': 'Florida',
...         'shortname': 'FL',
...         'info': {
...             'governor': 'Rick Scott'
...         }},
...         {'counties': [{'name': 'Dade', 'population': 12345},
...                         {'name': 'Broward', 'population': 40000},
...                         {'name': 'Palm Beach', 'population': 60000}]}],
...         {'state': 'Ohio',
...         'shortname': 'OH',
...         'info': {
...             'governor': 'John Kasich'
...         }},
...         {'counties': [{'name': 'Summit', 'population': 1234},
...                         {'name': 'Cuyahoga', 'population': 1337}]}]
>>> result = json_normalize(data, 'counties', ['state', 'shortname',
...                                             ['info', 'governor']])
>>> result
   name  population info.governor  state shortname
0    Dade    12345    Rick Scott  Florida      FL
1  Broward    40000    Rick Scott  Florida      FL
2  Palm Beach    60000    Rick Scott  Florida      FL
3    Summit     1234    John Kasich   Ohio      OH
4  Cuyahoga     1337    John Kasich   Ohio      OH
```

```
>>> data = {'A': [1, 2]}
>>> json_normalize(data, 'A', record_prefix='Prefix.')
   Prefix.0
0         1
1         2
```

### 34.1.5.3 pandas.io.json.build\_table\_schema

`pandas.io.json.build_table_schema` (*data*, *index=True*, *primary\_key=None*, *version=True*)

Create a Table schema from data.

#### Parameters

**data** [Series, DataFrame]

**index** : bool, default True

Whether to include `data.index` in the schema.

**primary\_key** : bool or None, default True

column names to designate as the primary key. The default *None* will set *'primaryKey'* to the index level or levels if the index is unique.

**version** : bool, default True

Whether to include a field *pandas\_version* with the version of pandas that generated the schema.

### Returns

**schema** [dict]

### Notes

See *\_as\_json\_table\_type* for conversion types. Timedeltas as converted to ISO8601 duration format with 9 decimal places after the seconds field for nanosecond precision.

Categoricals are converted to the *any* dtype, and use the *enum* field constraint to list the allowed values. The *ordered* attribute is included in an *ordered* field.

### Examples

```
>>> df = pd.DataFrame(
...     {'A': [1, 2, 3],
...      'B': ['a', 'b', 'c'],
...      'C': pd.date_range('2016-01-01', freq='d', periods=3),
...     }, index=pd.Index(range(3), name='idx'))
>>> build_table_schema(df)
{'fields': [{'name': 'idx', 'type': 'integer'},
{'name': 'A', 'type': 'integer'},
{'name': 'B', 'type': 'string'},
{'name': 'C', 'type': 'datetime'}],
'pandas_version': '0.20.0',
'primaryKey': ['idx']}
```

## 34.1.6 HTML

---

<code>read_html(io[, match, flavor, header, ...])</code>	Read HTML tables into a list of DataFrame objects.
--	--

---

### 34.1.6.1 pandas.read\_html

`pandas.read_html(io, match='.+', flavor=None, header=None, index_col=None, skiprows=None, attrs=None, parse_dates=False, tupleize_cols=None, thousands=',', encoding=None, decimal='.', converters=None, na_values=None, keep_default_na=True, displayed_only=True)`

Read HTML tables into a list of DataFrame objects.

**Parameters** **io** : str or file-like

A URL, a file-like object, or a raw string containing HTML. Note that lxml only accepts the http, ftp and file url protocols. If you have a URL that starts with 'https' you might try removing the 's'.

**match** : str or compiled regular expression, optional

The set of tables containing text matching this regex or string will be returned. Unless

the HTML is extremely simple you will probably need to pass a non-empty string here. Defaults to `‘.+’` (match any non-empty string). The default value will return all tables contained on a page. This value is converted to a regular expression so that there is consistent behavior between BeautifulSoup and lxml.

**flavor** : str or None, container of strings

The parsing engine to use. `‘bs4’` and `‘html5lib’` are synonymous with each other, they are both there for backwards compatibility. The default of `None` tries to use `lxml` to parse and if that fails it falls back on `bs4 + html5lib`.

**header** : int or list-like or None, optional

The row (or list of rows for a *MultiIndex*) to use to make the columns headers.

**index\_col** : int or list-like or None, optional

The column (or list of columns) to use to create the index.

**skiprows** : int or list-like or slice or None, optional

0-based. Number of rows to skip after parsing the column integer. If a sequence of integers or a slice is given, will skip the rows indexed by that sequence. Note that a single element sequence means ‘skip the nth row’ whereas an integer means ‘skip n rows’.

**attrs** : dict or None, optional

This is a dictionary of attributes that you can pass to use to identify the table in the HTML. These are not checked for validity before being passed to lxml or BeautifulSoup. However, these attributes must be valid HTML table attributes to work correctly. For example,

```
attrs = {'id': 'table'}
```

is a valid attribute dictionary because the `‘id’` HTML tag attribute is a valid HTML attribute for *any* HTML tag as per [this document](#).

```
attrs = {'asdf': 'table'}
```

is *not* a valid attribute dictionary because `‘asdf’` is not a valid HTML attribute even if it is a valid XML attribute. Valid HTML 4.01 table attributes can be found [here](#). A working draft of the HTML 5 spec can be found [here](#). It contains the latest information on table attributes for the modern web.

**parse\_dates** : bool, optional

See `read_csv()` for more details.

**tupleize\_cols** : bool, optional

If `False` try to parse multiple header rows into a *MultiIndex*, otherwise return raw tuples. Defaults to `False`.

Deprecated since version 0.21.0: This argument will be removed and will always convert to *MultiIndex*

**thousands** : str, optional

Separator to use to parse thousands. Defaults to `‘,’`.

**encoding** : str or None, optional

The encoding used to decode the web page. Defaults to `None`. “None” preserves the previous encoding behavior, which depends on the underlying parser library (e.g., the parser library will try to use the encoding provided by the document).

**decimal** : str, default ‘.’

Character to recognize as decimal point (e.g. use ‘,’ for European data).

New in version 0.19.0.

**converters** : dict, default None

Dict of functions for converting values in certain columns. Keys can either be integers or column labels, values are functions that take one input argument, the cell (not column) content, and return the transformed content.

New in version 0.19.0.

**na\_values** : iterable, default None

Custom NA values

New in version 0.19.0.

**keep\_default\_na** : bool, default True

If `na_values` are specified and `keep_default_na` is `False` the default NaN values are overridden, otherwise they’re appended to

New in version 0.19.0.

**display\_only** : bool, default True

Whether elements with “display: none” should be parsed

New in version 0.23.0.

### Returns

**dfs** [list of DataFrames]

### See also:

[`pandas.read\_csv`](#)

### Notes

Before using this function you should read the [gotchas about the HTML parsing libraries](#).

Expect to do some cleanup after you call this function. For example, you might need to manually assign column names if the column names are converted to NaN when you pass the `header=0` argument. We try to assume as little as possible about the structure of the table and push the idiosyncrasies of the HTML contained in the table to the user.

This function searches for `<table>` elements and only for `<tr>` and `<th>` rows and `<td>` elements within each `<tr>` or `<th>` element in the table. `<td>` stands for “table data”.

Similar to [`read\_csv\(\)`](#) the `header` argument is applied **after** `skiprows` is applied.

This function will *always* return a list of [`DataFrame`](#) or it will fail, e.g., it will *not* return an empty list.

## Examples

See the [read\\_html documentation in the IO section of the docs](#) for some examples of reading in HTML tables.

### 34.1.7 HDFStore: PyTables (HDF5)

<code>read_hdf(path_or_buf[, key, mode])</code>	Read from the store, close it if we opened it.
<code>HDFStore.put(key, value[, format, append])</code>	Store object in HDFStore
<code>HDFStore.append(key, value[, format, ...])</code>	Append to Table in file.
<code>HDFStore.get(key)</code>	Retrieve pandas object stored in file
<code>HDFStore.select(key[, where, start, stop, ...])</code>	Retrieve pandas object stored in file, optionally based on where criteria
<code>HDFStore.info()</code>	print detailed information on the store
<code>HDFStore.keys()</code>	Return a (potentially unordered) list of the keys corresponding to the objects stored in the HDFStore.

#### 34.1.7.1 pandas.read\_hdf

`pandas.read_hdf(path_or_buf, key=None, mode='r', **kwargs)`

Read from the store, close it if we opened it.

Retrieve pandas object stored in file, optionally based on where criteria

**Parameters** `path_or_buf` : string, buffer or path object

Path to the file to open, or an open `pandas.HDFStore` object. Supports any object implementing the `__fspath__` protocol. This includes `pathlib.Path` and `py._path.local.LocalPath` objects.

New in version 0.19.0: support for `pathlib`, `py.path`.

New in version 0.21.0: support for `__fspath__` proptocol.

**key** : object, optional

The group identifier in the store. Can be omitted if the HDF file contains a single pandas object.

**mode** : {'r', 'r+', 'a'}, optional

Mode to use when opening the file. Ignored if `path_or_buf` is a `pandas.HDFStore`. Default is 'r'.

**where** : list, optional

A list of Term (or convertible) objects.

**start** : int, optional

Row number to start selection.

**stop** : int, optional

Row number to stop selection.

**columns** : list, optional

A list of columns names to return.

**iterator** : bool, optional

Return an iterator object.

**chunksize** : int, optional

Number of rows to include in an iteration when using an iterator.

**errors** : str, default 'strict'

Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

**\*\*kwargs**

Additional keyword arguments passed to HDFStore.

**Returns** **item** : object

The selected object. Return type depends on the object stored.

**See also:**

`pandas.DataFrame.to_hdf` write a HDF file from a DataFrame

`pandas.HDFStore` low-level access to HDF files

## Examples

```
>>> df = pd.DataFrame([[1, 1.0, 'a']], columns=['x', 'y', 'z'])
>>> df.to_hdf('./store.h5', 'data')
>>> reread = pd.read_hdf('./store.h5')
```

### 34.1.7.2 pandas.HDFStore.put

`HDFStore.put` (*key, value, format=None, append=False, \*\*kwargs*)  
Store object in HDFStore

#### Parameters

**key** [object]

**value** [{Series, DataFrame, Panel}]

**format** : 'fixed(f)|table(t)', default is 'fixed'

**fixed(f)** [Fixed format] Fast writing/reading. Not-appendable, nor searchable

**table(t)** [Table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

**append** : boolean, default False

This will force Table format, append the input data to the existing.

**data\_columns** : list of columns to create as data columns, or True to use all columns. See [here](#) # noqa

**encoding** [default None, provide an encoding for strings]

**dropna** : boolean, default False, do not write an ALL nan row to the store settable by the option 'io.hdf.dropna\_table'

### 34.1.7.3 pandas.HDFStore.append

`HDFStore.append(key, value, format=None, append=True, columns=None, dropna=None, **kwargs)`  
Append to Table in file. Node must already exist and be Table format.

#### Parameters

**key** [object]

**value** [{Series, DataFrame, Panel}]

**format: ‘table’ is the default**

**table(t)** [table format] Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data

**append** : boolean, default True, append the input data to the existing

**data\_columns** : list of columns, or True, default None

List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See [here](#).

**min\_itemsize** [dict of columns that specify minimum string sizes]

**nan\_rep** [string to use as string nan representation]

**chunksize** [size to chunk the writing]

**expectedrows** [expected TOTAL row size of this table]

**encoding** [default None, provide an encoding for strings]

**dropna** : boolean, default False, do not write an ALL nan row to the store settable by the option ‘io.hdf.dropna\_table’

#### Notes

Does *not* check if data being appended overlaps with existing data in the table, so be careful

### 34.1.7.4 pandas.HDFStore.get

`HDFStore.get(key)`  
Retrieve pandas object stored in file

#### Parameters

**key** [object]

#### Returns

**obj** [type of object stored in file]



### 34.1.7.5 pandas.HDFStore.select

`HDFStore.select` (*key*, *where=None*, *start=None*, *stop=None*, *columns=None*, *iterator=False*, *chunk-size=None*, *auto\_close=False*, *\*\*kwargs*)

Retrieve pandas object stored in file, optionally based on where criteria

#### Parameters

**key** [object]

**where** [list of Term (or convertible) objects, optional]

**start** [integer (defaults to None), row number to start selection]

**stop** [integer (defaults to None), row number to stop selection]

**columns** : a list of columns that if not None, will limit the return columns

**iterator** [boolean, return an iterator, default False]

**chunksize** [nrows to include in iteration, return an iterator]

**auto\_close** : boolean, should automatically close the store when finished, default is False

#### Returns

The selected object

### 34.1.7.6 pandas.HDFStore.info

`HDFStore.info` ()

print detailed information on the store

New in version 0.21.0.

### 34.1.7.7 pandas.HDFStore.keys

`HDFStore.keys` ()

Return a (potentially unordered) list of the keys corresponding to the objects stored in the HDFStore. These are ABSOLUTE path-names (e.g. have the leading '/')

## 34.1.8 Feather

---

`read_feather`(*path*[, *nthreads*])

Load a feather-format object from the file path

---

### 34.1.8.1 pandas.read\_feather

`pandas.read_feather` (*path*, *nthreads=1*)

Load a feather-format object from the file path

#### Parameters

**path** [string file path, or file-like object]

**nthreads** : int, default 1

Number of CPU threads to use when reading to pandas.DataFrame

**Returns**

type of object stored in file

### 34.1.9 Parquet

---

<code>read_parquet(path[, engine, columns])</code>	Load a parquet object from the file path, returning a DataFrame.
--	--

---

#### 34.1.9.1 pandas.read\_parquet

`pandas.read_parquet(path, engine='auto', columns=None, **kwargs)`

Load a parquet object from the file path, returning a DataFrame.

**Parameters** **path** : string

File path

**columns**: list, default=None

If not None, only these columns will be read from the file.

**engine** : { 'auto', 'pyarrow', 'fastparquet' }, default 'auto'

Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

**kwargs** are passed to the engine

**Returns**

DataFrame

### 34.1.10 SAS

---

<code>read_sas(filepath_or_buffer[, format, ...])</code>	Read SAS files stored as either XPORT or SAS7BDAT format files.
--	---

---

#### 34.1.10.1 pandas.read\_sas

`pandas.read_sas(filepath_or_buffer, format=None, index=None, encoding=None, chunksize=None, iterator=False)`

Read SAS files stored as either XPORT or SAS7BDAT format files.

**Parameters** **filepath\_or\_buffer** : string or file-like object

Path to the SAS file.

**format** : string { 'xport', 'sas7bdat' } or None

If None, file format is inferred. If 'xport' or 'sas7bdat', uses the corresponding format.

**index** : identifier of index column, defaults to None

Identifier of column that should be used as index of the DataFrame.

**encoding** : string, default is None

Encoding for text data. If None, text data are stored as raw bytes.

**chunksize** : int

Read file *chunksize* lines at a time, returns iterator.

**iterator** : bool, defaults to False

If True, returns an iterator for reading the file incrementally.

#### Returns

**DataFrame** if **iterator=False** and **chunksize=None**, else **SAS7BDATReader**  
or **XportReader**

### 34.1.11 SQL

<code>read_sql_table(table_name, con[, schema, ...])</code>	Read SQL database table into a DataFrame.
<code>read_sql_query(sql, con[, index_col, ...])</code>	Read SQL query into a DataFrame.
<code>read_sql(sql, con[, index_col, ...])</code>	Read SQL query or database table into a DataFrame.

### 34.1.12 Google BigQuery

<code>read_gbq(query[, project_id, index_col, ...])</code>	Load data from Google BigQuery.
--	---------------------------------

#### 34.1.12.1 pandas.read\_gbq

`pandas.read_gbq(query, project_id=None, index_col=None, col_order=None, reauth=False, verbose=None, private_key=None, dialect='legacy', **kwargs)`  
Load data from Google BigQuery.

This function requires the [pandas-gbq](#) package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If “private\_key” is not provided:

By default “application default credentials” are used.

If default application credentials are not found or are restrictive, user account credentials are used. In this case, you will be asked to grant permissions for product name ‘pandas GBQ’.

- If “private\_key” is provided:

Service account credentials will be used to authenticate.

**Parameters** **query** : str

SQL-Like Query to return data values.

**project\_id** : str

Google BigQuery Account project ID.

**index\_col** : str, optional

Name of result column to use for index in results DataFrame.

**col\_order** : list(str), optional

List of BigQuery column names in the desired order for results DataFrame.

**reauth** : boolean, default False

Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

**private\_key** : str, optional

Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

**dialect** : str, default 'legacy'

SQL syntax dialect to use. Value can be one of:

'**legacy**' Use BigQuery's legacy SQL dialect. For more information see [BigQuery Legacy SQL Reference](#).

'**standard**' Use BigQuery's standard SQL, which is compliant with the SQL 2011 standard. For more information see [BigQuery Standard SQL Reference](#).

**verbose** : boolean, deprecated

*Deprecated in Pandas-GBQ 0.4.0.* Use the [logging module](#) to adjust verbosity instead.

**kwargs** : dict

Arbitrary keyword arguments. configuration (dict): query config parameters for job processing. For example:

```
configuration = {'query': {'useQueryCache': False}}
```

For more information see [BigQuery SQL Reference](#)

**Returns df: DataFrame**

DataFrame representing results of query.

See also:

[`pandas\_gbq.read\_gbq`](#) This function in the pandas-gbq library.

[`pandas.DataFrame.to\_gbq`](#) Write a DataFrame to Google BigQuery.

### 34.1.13 STATA

---

<code>read_stata(filepath_or_buffer[, ...])</code>	Read Stata file into DataFrame.
--	---------------------------------

---

#### 34.1.13.1 pandas.read\_stata

`pandas.read_stata(filepath_or_buffer, convert_dates=True, convert_categoricals=True, encoding=None, index_col=None, convert_missing=False, preserve_dtypes=True, columns=None, order_categoricals=True, chunksize=None, iterator=False)`

Read Stata file into DataFrame.

**Parameters filepath\_or\_buffer** : string or file-like object

Path to .dta file or object implementing a binary read() functions.

**convert\_dates** : boolean, defaults to True

Convert date variables to DataFrame time values.

**convert\_categoricals** : boolean, defaults to True

Read value labels and convert columns to Categorical/Factor variables.

**encoding** : string, None or encoding

Encoding used to parse the files. None defaults to latin-1.

**index\_col** : string, optional, default: None

Column to set as index.

**convert\_missing** : boolean, defaults to False

Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nan. If True, columns containing missing values are returned with object data types and missing values are represented by StataMissingValue objects.

**preserve\_dtypes** : boolean, defaults to True

Preserve Stata datatypes. If False, numeric data are upcast to pandas default types for foreign data (float64 or int64).

**columns** : list or None

Columns to retain. Columns will be returned in the given order. None returns all columns.

**order\_categoricals** : boolean, defaults to True

Flag indicating whether converted categorical data are ordered.

**chunksize** : int, default None

Return StataReader object for iterations, returns chunks with given number of lines.

**iterator** : boolean, default False

Return StataReader object.

## Returns

**DataFrame or StataReader**

See also:

**pandas.io.stata.StataReader** low-level reader for Stata data files

[\*pandas.DataFrame.to\\_stata\*](#) export Stata data files

## Examples

Read a Stata dta file:

```
>>> import pandas as pd
>>> df = pd.read_stata('filename.dta')
```

Read a Stata dta file in 10,000 line chunks:

```
>>> itr = pd.read_stata('filename.dta', chunksize=10000)
>>> for chunk in itr:
...     do_something(chunk)
```

<code>StataReader.data(**kwargs)</code>	(DEPRECATED) Reads observations from Stata file, converting them into a dataframe
<code>StataReader.data_label()</code>	Returns data label of Stata file
<code>StataReader.value_labels()</code>	Returns a dict, associating each variable name a dict, associating each value its corresponding label
<code>StataReader.variable_labels()</code>	Returns variable labels as a dict, associating each variable name with corresponding label
<code>StataWriter.write_file()</code>	

### 34.1.13.2 pandas.io.stata.StataReader.data

`StataReader.data(**kwargs)`

Reads observations from Stata file, converting them into a dataframe

Deprecated since version This: is a legacy method. Use *read* in new code.

**Parameters** `convert_dates` : boolean, defaults to True

Convert date variables to DataFrame time values.

`convert_categoricals` : boolean, defaults to True

Read value labels and convert columns to Categorical/Factor variables.

`index_col` : string, optional, default: None

Column to set as index.

`convert_missing` : boolean, defaults to False

Flag indicating whether to convert missing values to their Stata representations. If False, missing values are replaced with nan. If True, columns containing missing values are returned with object data types and missing values are represented by `StataMissingValue` objects.

`preserve_dtypes` : boolean, defaults to True

Preserve Stata datatypes. If False, numeric data are upcast to pandas default types for foreign data (float64 or int64).

`columns` : list or None

Columns to retain. Columns will be returned in the given order. None returns all columns.

`order_categoricals` : boolean, defaults to True

Flag indicating whether converted categorical data are ordered.

**Returns**

**DataFrame**

### 34.1.13.3 pandas.io.stata.StataReader.data\_label

`StataReader.data_label()`  
Returns data label of Stata file

### 34.1.13.4 pandas.io.stata.StataReader.value\_labels

`StataReader.value_labels()`  
Returns a dict, associating each variable name a dict, associating each value its corresponding label

### 34.1.13.5 pandas.io.stata.StataReader.variable\_labels

`StataReader.variable_labels()`  
Returns variable labels as a dict, associating each variable name with corresponding label

### 34.1.13.6 pandas.io.stata.StataWriter.write\_file

`StataWriter.write_file()`

## 34.2 General functions

### 34.2.1 Data manipulations

<code>melt(frame[, id_vars, value_vars, var_name, ...])</code>	“Unpivots” a DataFrame from wide format to long format, optionally leaving identifier variables set.
<code>pivot(index, columns, values)</code>	Produce ‘pivot’ table based on 3 columns of this DataFrame.
<code>pivot_table(data[, values, index, columns, ...])</code>	Create a spreadsheet-style pivot table as a DataFrame.
<code>crosstab(index, columns[, values, rownames, ...])</code>	Compute a simple cross-tabulation of two (or more) factors.
<code>cut(x, bins[, right, labels, retbins, ...])</code>	Bin values into discrete intervals.
<code>qcut(x, q[, labels, retbins, precision, ...])</code>	Quantile-based discretization function.
<code>merge(left, right[, how, on, left_on, ...])</code>	Merge DataFrame objects by performing a database-style join operation by columns or indexes.
<code>merge_ordered(left, right[, on, left_on, ...])</code>	Perform merge with optional filling/interpolation designed for ordered data like time series data.
<code>merge_asof(left, right[, on, left_on, ...])</code>	Perform an asof merge.
<code>concat(objs[, axis, join, join_axes, ...])</code>	Concatenate pandas objects along a particular axis with optional set logic along the other axes.
<code>get_dummies(data[, prefix, prefix_sep, ...])</code>	Convert categorical variable into dummy/indicator variables
<code>factorize(values[, sort, order, ...])</code>	Encode the object as an enumerated type or categorical variable.
<code>unique(values)</code>	Hash table-based unique.
<code>wide_to_long(df, stubnames, i, j[, sep, suffix])</code>	Wide panel to long format.

### 34.2.1.1 pandas.melt

`pandas.melt` (*frame*, *id\_vars=None*, *value\_vars=None*, *var\_name=None*, *value\_name='value'*, *col\_level=None*)

“Unpivots” a DataFrame from wide format to long format, optionally leaving identifier variables set.

This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id\_vars*), while all other columns, considered measured variables (*value\_vars*), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

#### Parameters

**frame** [DataFrame]

**id\_vars** : tuple, list, or ndarray, optional

Column(s) to use as identifier variables.

**value\_vars** : tuple, list, or ndarray, optional

Column(s) to unpivot. If not specified, uses all columns that are not set as *id\_vars*.

**var\_name** : scalar

Name to use for the ‘variable’ column. If None it uses `frame.columns.name` or ‘variable’.

**value\_name** : scalar, default ‘value’

Name to use for the ‘value’ column.

**col\_level** : int or string, optional

If columns are a MultiIndex then use this level to melt.

See also:

`DataFrame.melt`, `pivot_table`, `DataFrame.pivot`

#### Examples

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                   'B': {0: 1, 1: 3, 2: 5},
...                   'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a         B       1
1  b         B       3
2  c         B       5
```

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a         B       1
```

(continues on next page)



(continued from previous page)

1	b	B	3
2	c	B	5
3	a	C	2
4	b	C	4
5	c	C	6

The names of 'variable' and 'value' columns can be customized:

```
>>> pd.melt(df, id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
  A myVarname myValname
0  a         B         1
1  b         B         3
2  c         B         5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A B C
   D E F
0  a 1 2
1  b 3 4
2  c 5 6
```

```
>>> pd.melt(df, col_level=0, id_vars=['A'], value_vars=['B'])
  A variable  value
0  a         B     1
1  b         B     3
2  c         B     5
```

```
>>> pd.melt(df, id_vars=[('A', 'D')], value_vars=[('B', 'E')])
  (A, D) variable_0 variable_1  value
0      a         B         E     1
1      b         B         E     3
2      c         B         E     5
```

### 34.2.1.2 pandas.pivot

pandas.**pivot** (*index, columns, values*)

Produce 'pivot' table based on 3 columns of this DataFrame. Uses unique values from index / columns and fills with values.

**Parameters** **index** : ndarray

Labels to use to make new frame's index

**columns** : ndarray

Labels to use to make new frame's columns

**values** : ndarray

Values to use for populating new frame's values

**Returns**

**DataFrame**

See also:

`DataFrame.pivot_table` generalization of pivot that can handle duplicate values for one index/column pair

## Notes

Obviously, all 3 of the input arguments must have the same length

### 34.2.1.3 pandas.pivot\_table

`pandas.pivot_table` (*data*, *values=None*, *index=None*, *columns=None*, *aggfunc='mean'*, *fill\_value=None*, *margins=False*, *dropna=True*, *margins\_name='All'*)

Create a spreadsheet-style pivot table as a DataFrame. The levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame

#### Parameters

**data** [DataFrame]

**values** [column to aggregate, optional]

**index** : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

**columns** : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

**aggfunc** : function, list of functions, dict, default `numpy.mean`

If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

**fill\_value** : scalar, default `None`

Value to replace missing values with

**margins** : boolean, default `False`

Add all row / columns (e.g. for subtotal / grand totals)

**dropna** : boolean, default `True`

Do not include columns whose entries are all NaN

**margins\_name** : string, default `'All'`

Name of the row / column that will contain the totals when margins is `True`.

#### Returns

**table** [DataFrame]

See also:

`DataFrame.pivot` pivot without aggregation that can handle non-numeric data

## Examples

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                           "bar", "bar", "bar", "bar"],
...                     "B": ["one", "one", "one", "two", "two",
...                           "one", "one", "two", "two"],
...                     "C": ["small", "large", "large", "small",
...                           "small", "large", "small", "small",
...                           "large"],
...                     "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
   A  B  C  D
0  foo one small 1
1  foo one large 2
2  foo one large 2
3  foo two small 3
4  foo two small 3
5  bar one large 4
6  bar one small 5
7  bar two small 6
8  bar two large 7
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                       columns=['C'], aggfunc=np.sum)
>>> table
C      large  small
A  B
bar one    4.0    5.0
   two    7.0    6.0
foo one    4.0    1.0
   two    NaN    6.0
```

```
>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                       columns=['C'], aggfunc=np.sum)
>>> table
C      large  small
A  B
bar one    4.0    5.0
   two    7.0    6.0
foo one    4.0    1.0
   two    NaN    6.0
```

```
>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                       aggfunc={'D': np.mean,
...                                  'E': [min, max, np.mean]})
>>> table
      D      E
      mean max median min
A  C
bar large  5.500000  16   14.5  13
   small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
   small  2.333333  12   11.0   8
```

### 34.2.1.4 pandas.crosstab

`pandas.crosstab` (*index*, *columns*, *values=None*, *rownames=None*, *colnames=None*, *aggfunc=None*, *margins=False*, *margins\_name='All'*, *dropna=True*, *normalize=False*)

Compute a simple cross-tabulation of two (or more) factors. By default computes a frequency table of the factors unless an array of values and an aggregation function are passed

**Parameters** **index** : array-like, Series, or list of arrays/Series

Values to group by in the rows

**columns** : array-like, Series, or list of arrays/Series

Values to group by in the columns

**values** : array-like, optional

Array of values to aggregate according to the factors. Requires *aggfunc* be specified.

**aggfunc** : function, optional

If specified, requires *values* be specified as well

**rownames** : sequence, default None

If passed, must match number of row arrays passed

**colnames** : sequence, default None

If passed, must match number of column arrays passed

**margins** : boolean, default False

Add row/column margins (subtotals)

**margins\_name** : string, default 'All'

Name of the row / column that will contain the totals when margins is True.

New in version 0.21.0.

**dropna** : boolean, default True

Do not include columns whose entries are all NaN

**normalize** : boolean, {'all', 'index', 'columns'}, or {0,1}, default False

Normalize by dividing all values by the sum of values.

- If passed 'all' or *True*, will normalize over all values.
- If passed 'index' will normalize over each row.
- If passed 'columns' will normalize over each column.
- If margins is *True*, will also normalize margin values.

New in version 0.18.1.

**Returns**

**crosstab** [DataFrame]

## Notes

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified.

Any input passed containing Categorical data will have **all** of its categories included in the cross-tabulation, even if the actual data does not contain any instances of a particular category.

In the event that there aren't overlapping indexes an empty DataFrame will be returned.

## Examples

```
>>> a = np.array(["foo", "foo", "foo", "foo", "bar", "bar",
...              "bar", "bar", "foo", "foo", "foo"], dtype=object)
>>> b = np.array(["one", "one", "one", "two", "one", "one",
...              "one", "two", "two", "two", "one"], dtype=object)
>>> c = np.array(["dull", "dull", "shiny", "dull", "dull", "shiny",
...              "shiny", "dull", "shiny", "shiny", "shiny"],
...              dtype=object)
```

```
>>> pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
...
b    one      two
c    dull shiny dull shiny
a
bar    1      2    1      0
foo    2      2    1      2
```

```
>>> foo = pd.Categorical(['a', 'b'], categories=['a', 'b', 'c'])
>>> bar = pd.Categorical(['d', 'e'], categories=['d', 'e', 'f'])
>>> crosstab(foo, bar) # 'c' and 'f' are not represented in the data,
...                   # but they still will be counted in the output
...
col_0  d  e  f
row_0
a      1  0  0
b      0  1  0
c      0  0  0
```

### 34.2.1.5 pandas.cut

`pandas.cut` (*x*, *bins*, *right=True*, *labels=None*, *retbins=False*, *precision=3*, *include\_lowest=False*, *duplicates='raise'*)

Bin values into discrete intervals.

Use *cut* when you need to segment and sort data values into bins. This function is also useful for going from a continuous variable to a categorical variable. For example, *cut* could convert ages to groups of age ranges. Supports binning into an equal number of bins, or a pre-specified array of bins.

**Parameters** *x* : array-like

The input array to be binned. Must be 1-dimensional.

**bins** : int, sequence of scalars, or `pandas.IntervalIndex`

The criteria to bin by.

- **int** : Defines the number of equal-width bins in the range of *x*. The range of *x* is extended by .1% on each side to include the minimum and maximum values of *x*.
- **sequence of scalars** : Defines the bin edges allowing for non-uniform width. No extension of the range of *x* is done.
- **IntervalIndex** : Defines the exact bins to be used.

**right** : bool, default True

Indicates whether *bins* includes the rightmost edge or not. If `right == True` (the default), then the *bins* `[1, 2, 3, 4]` indicate (1,2], (2,3], (3,4]. This argument is ignored when *bins* is an *IntervalIndex*.

**labels** : array or bool, optional

Specifies the labels for the returned bins. Must be the same length as the resulting bins. If False, returns only integer indicators of the bins. This affects the type of the output container (see below). This argument is ignored when *bins* is an *IntervalIndex*.

**retbins** : bool, default False

Whether to return the bins or not. Useful when bins is provided as a scalar.

**precision** : int, default 3

The precision at which to store and display the bins labels.

**include\_lowest** : bool, default False

Whether the first interval should be left-inclusive or not.

**duplicates** : {default 'raise', 'drop'}, optional

If bin edges are not unique, raise *ValueError* or drop non-uniques.

New in version 0.23.0.

**Returns out** : pandas.Categorical, Series, or ndarray

An array-like object representing the respective bin for each value of *x*. The type depends on the value of *labels*.

- True (default) : returns a Series for Series *x* or a pandas.Categorical for all other inputs. The values stored within are Interval dtype.
- sequence of scalars : returns a Series for Series *x* or a pandas.Categorical for all other inputs. The values stored within are whatever the type in the sequence is.
- False : returns an ndarray of integers.

**bins** : numpy.ndarray or IntervalIndex.

The computed or specified bins. Only returned when *retbins=True*. For scalar or sequence *bins*, this is an ndarray with the computed bins. If set *duplicates=drop*, *bins* will drop non-unique bin. For an *IntervalIndex bins*, this is equal to *bins*.

**See also:**

**qcut** Discretize variable into equal-sized buckets based on rank or based on sample quantiles.

**pandas.Categorical** Array type for storing data that come from a fixed set of values.

**Series** One-dimensional array with axis labels (including time series).

**pandas.IntervalIndex** Immutable Index implementing an ordered, sliceable set.

## Notes

Any NA values will be NA in the result. Out of bounds values will be NA in the resulting Series or pandas.Categorical object.

## Examples

Discretize into three equal-sized bins.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3)
...
[(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] ...
```

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]), 3, retbins=True)
...
([(0.994, 3.0], (5.0, 7.0], (3.0, 5.0], (3.0, 5.0], (5.0, 7.0], ...
Categories (3, interval[float64]): [(0.994, 3.0] < (3.0, 5.0] ...
array([0.994, 3.    , 5.    , 7.    ]))
```

Discovers the same bins, but assign them specific labels. Notice that the returned Categorical's categories are *labels* and is ordered.

```
>>> pd.cut(np.array([1, 7, 5, 4, 6, 3]),
...         3, labels=["bad", "medium", "good"])
[bad, good, medium, medium, good, bad]
Categories (3, object): [bad < medium < good]
```

labels=False implies you just want the bins back.

```
>>> pd.cut([0, 1, 1, 2], bins=4, labels=False)
array([0, 1, 1, 3])
```

Passing a Series as an input returns a Series with categorical dtype:

```
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...               index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, 3)
...
a    (1.992, 4.667]
b    (1.992, 4.667]
c    (4.667, 7.333]
d    (7.333, 10.0]
e    (7.333, 10.0]
dtype: category
Categories (3, interval[float64]): [(1.992, 4.667] < (4.667, ...
```

Passing a Series as an input returns a Series with mapping value. It is used to map numerically to intervals based on bins.

```
>>> s = pd.Series(np.array([2, 4, 6, 8, 10]),
...               index=['a', 'b', 'c', 'd', 'e'])
>>> pd.cut(s, [0, 2, 4, 6, 8, 10], labels=False, retbins=True, right=False)
...
(a    0.0
```

(continues on next page)

(continued from previous page)

```
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64, array([0, 2, 4, 6, 8]))
```

Use *drop* optional when bins is not unique

```
>>> pd.cut(s, [0, 2, 4, 6, 10, 10], labels=False, retbins=True,
...        right=False, duplicates='drop')
...
(a    0.0
b    1.0
c    2.0
d    3.0
e    3.0
dtype: float64, array([0, 2, 4, 6, 8]))
```

Passing an `IntervalIndex` for *bins* results in those categories exactly. Notice that values not covered by the `IntervalIndex` are set to `NaN`. 0 is to the left of the first bin (which is closed on the right), and 1.5 falls between two bins.

```
>>> bins = pd.IntervalIndex.from_tuples([(0, 1), (2, 3), (4, 5)])
>>> pd.cut([0, 0.5, 1.5, 2.5, 4.5], bins)
[NaN, (0, 1], NaN, (2, 3], (4, 5]]
Categories (3, interval[int64]): [(0, 1] < (2, 3] < (4, 5]]
```

### 34.2.1.6 pandas.qcut

`pandas.qcut(x, q, labels=None, retbins=False, precision=3, duplicates='raise')`

Quantile-based discretization function. Discretize variable into equal-sized buckets based on rank or based on sample quantiles. For example 1000 values for 10 quantiles would produce a Categorical object indicating quantile membership for each data point.

#### Parameters

**x** [1d ndarray or Series]

**q** : integer or array of quantiles

Number of quantiles. 10 for deciles, 4 for quartiles, etc. Alternately array of quantiles, e.g. [0, .25, .5, .75, 1.] for quartiles

**labels** : array or boolean, default None

Used as labels for the resulting bins. Must be of the same length as the resulting bins. If False, return only integer indicators of the bins.

**retbins** : bool, optional

Whether to return the (bins, labels) or not. Can be useful if bins is given as a scalar.

**precision** : int, optional

The precision at which to store and display the bins labels

**duplicates** : {default 'raise', 'drop'}, optional



If bin edges are not unique, raise ValueError or drop non-uniques.

New in version 0.20.0.

**Returns out** : Categorical or Series or array of integers if labels is False

The return type (Categorical or Series) depends on the input: a Series of type category if input is a Series else Categorical. Bins are represented as categories when categorical data is returned.

**bins** : ndarray of floats

Returned only if *retbins* is True.

## Notes

Out of bounds values will be NA in the resulting Categorical object

## Examples

```
>>> pd.qcut(range(5), 4)
...
[(-0.001, 1.0], (-0.001, 1.0], (1.0, 2.0], (2.0, 3.0], (3.0, 4.0]]
Categories (4, interval[float64]): [(-0.001, 1.0] < (1.0, 2.0] ...
```

```
>>> pd.qcut(range(5), 3, labels=["good", "medium", "bad"])
...
[good, good, medium, bad, bad]
Categories (3, object): [good < medium < bad]
```

```
>>> pd.qcut(range(5), 4, labels=False)
array([0, 0, 1, 2, 3])
```

### 34.2.1.7 pandas.merge

`pandas.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indicator=False, validate=None)`

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

#### Parameters

**left** [DataFrame]

**right** [DataFrame]

**how** : {'left', 'right', 'outer', 'inner'}, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically

- **inner**: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

**on** : label or list

Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

**left\_on** : label or list, or array-like

Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

**right\_on** : label or list, or array-like

Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

**left\_index** : boolean, default False

Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

**right\_index** : boolean, default False

Use the index from the right DataFrame as the join key. Same caveats as left\_index

**sort** : boolean, default False

Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

**suffixes** : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

**copy** : boolean, default True

If False, do not copy data unnecessarily

**indicator** : boolean or string, default False

If True, adds a column to output DataFrame called “\_merge” with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of “left\_only” for observations whose merge key only appears in ‘left’ DataFrame, “right\_only” for observations whose merge key only appears in ‘right’ DataFrame, and “both” if the observation’s merge key is found in both.

**validate** : string, default None

If specified, checks if merge is of specified type.

- “one\_to\_one” or “1:1”: check if merge keys are unique in both left and right datasets.
- “one\_to\_many” or “1:m”: check if merge keys are unique in left dataset.
- “many\_to\_one” or “m:1”: check if merge keys are unique in right dataset.
- “many\_to\_many” or “m:m”: allowed, but does not result in checks.

New in version 0.21.0.

**Returns** `merged` : DataFrame

The output type will be the same as 'left', if it is a subclass of DataFrame.

**See also:**

`merge_ordered`, `merge_asof`, `DataFrame.join`

## Notes

Support for specifying index levels as the `on`, `left_on`, and `right_on` parameters was added in version 0.23.0

## Examples

```
>>> A          >>> B
   lkey value    rkey value
0   foo    1      0   foo    5
1   bar    2      1   bar    6
2   baz    3      2   qux    7
3   foo    4      3   bar    8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0   foo      1    foo      5
1   foo      4    foo      5
2   bar      2    bar      6
3   bar      2    bar      8
4   baz      3    NaN     NaN
5   NaN     NaN    qux      7
```

### 34.2.1.8 pandas.merge\_ordered

`pandas.merge_ordered(left, right, on=None, left_on=None, right_on=None, left_by=None, right_by=None, fill_method=None, suffixes=('_x', '_y'), how='outer')`

Perform merge with optional filling/interpolation designed for ordered data like time series data. Optionally perform group-wise merge (see examples)

#### Parameters

**left** [DataFrame]

**right** [DataFrame]

**on** : label or list

Field names to join on. Must be found in both DataFrames.

**left\_on** : label or list, or array-like

Field names to join on in left DataFrame. Can be a vector or list of vectors of the length of the DataFrame to use a particular vector as the join key instead of columns

**right\_on** : label or list, or array-like

Field names to join on in right DataFrame or vector/list of vectors per left\_on docs

**left\_by** : column name or list of column names

Group left DataFrame by group columns and merge piece by piece with right DataFrame

**right\_by** : column name or list of column names

Group right DataFrame by group columns and merge piece by piece with left DataFrame

**fill\_method** : { 'ffill', None }, default None

Interpolation method for data

**suffixes** : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

**how** : { 'left', 'right', 'outer', 'inner' }, default 'outer'

- left: use only keys from left frame (SQL: left outer join)
- right: use only keys from right frame (SQL: right outer join)
- outer: use union of keys from both frames (SQL: full outer join)
- inner: use intersection of keys from both frames (SQL: inner join)

New in version 0.19.0.

**Returns** **merged** : DataFrame

The output type will be the same as 'left', if it is a subclass of DataFrame.

**See also:**

*merge, merge\_asof*

## Examples

```
>>> A
   key  lvalue group
0    a      1     a
1    c      2     a
2    e      3     a
3    a      1     b
4    c      2     b
5    e      3     b

>>> B
   key  rvalue
0    b      1
1    c      2
2    d      3
```

```
>>> merge_ordered(A, B, fill_method='ffill', left_by='group')
   group key  lvalue  rvalue
0     a   a      1     NaN
1     a   b      1     1.0
2     a   c      2     2.0
3     a   d      2     3.0
4     a   e      3     3.0
5     b   a      1     NaN
6     b   b      1     1.0
7     b   c      2     2.0
8     b   d      2     3.0
9     b   e      3     3.0
```

### 34.2.1.9 pandas.merge\_asof

`pandas.merge_asof` (*left*, *right*, *on=None*, *left\_on=None*, *right\_on=None*, *left\_index=False*, *right\_index=False*, *by=None*, *left\_by=None*, *right\_by=None*, *suffixes=('\_x', '\_y')*, *tolerance=None*, *allow\_exact\_matches=True*, *direction='backward'*)

Perform an asof merge. This is similar to a left-join except that we match on nearest key rather than equal keys.

Both DataFrames must be sorted by the key.

For each row in the left DataFrame:

- A “backward” search selects the last row in the right DataFrame whose ‘on’ key is less than or equal to the left’s key.
- A “forward” search selects the first row in the right DataFrame whose ‘on’ key is greater than or equal to the left’s key.
- A “nearest” search selects the row in the right DataFrame whose ‘on’ key is closest in absolute distance to the left’s key.

The default is “backward” and is compatible in versions below 0.20.0. The direction parameter was added in version 0.20.0 and introduces “forward” and “nearest”.

Optionally match on equivalent keys with ‘by’ before searching with ‘on’.

New in version 0.19.0.

#### Parameters

**left** [DataFrame]

**right** [DataFrame]

**on** : label

Field name to join on. Must be found in both DataFrames. The data MUST be ordered. Furthermore this must be a numeric column, such as datetimelike, integer, or float. On or left\_on/right\_on must be given.

**left\_on** : label

Field name to join on in left DataFrame.

**right\_on** : label

Field name to join on in right DataFrame.

**left\_index** : boolean

Use the index of the left DataFrame as the join key.

New in version 0.19.2.

**right\_index** : boolean

Use the index of the right DataFrame as the join key.

New in version 0.19.2.

**by** : column name or list of column names

Match on these columns before performing merge operation.

**left\_by** : column name

Field names to match on in the left DataFrame.

New in version 0.19.2.

**right\_by** : column name

Field names to match on in the right DataFrame.

New in version 0.19.2.

**suffixes** : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively.

**tolerance** : integer or Timedelta, optional, default None

Select asof tolerance within this range; must be compatible with the merge index.

**allow\_exact\_matches** : boolean, default True

- If True, allow matching with the same 'on' value (i.e. less-than-or-equal-to / greater-than-or-equal-to)
- If False, don't match the same 'on' value (i.e., strictly less-than / strictly greater-than)

**direction** : 'backward' (default), 'forward', or 'nearest'

Whether to search for prior, subsequent, or closest matches.

New in version 0.20.0.

### Returns

**merged** [DataFrame]

**See also:**

[\*merge\*](#), [\*merge\\_ordered\*](#)

### Examples

```
>>> left = pd.DataFrame({'a': [1, 5, 10], 'left_val': ['a', 'b', 'c']})
>>> left
   a left_val
0  1         a
1  5         b
2 10         c
```

```
>>> right = pd.DataFrame({'a': [1, 2, 3, 6, 7],
...                       'right_val': [1, 2, 3, 6, 7]})
>>> right
   a right_val
0  1          1
1  2          2
2  3          3
3  6          6
4  7          7
```

```
>>> pd.merge_asof(left, right, on='a')
   a left_val right_val
0  1         a         1
1  5         b         3
2 10         c         7
```

```
>>> pd.merge_asof(left, right, on='a', allow_exact_matches=False)
   a left_val right_val
0  1         a        NaN
1  5         b         3.0
2 10         c         7.0
```

```
>>> pd.merge_asof(left, right, on='a', direction='forward')
   a left_val right_val
0  1         a         1.0
1  5         b         6.0
2 10         c         NaN
```

```
>>> pd.merge_asof(left, right, on='a', direction='nearest')
   a left_val right_val
0  1         a         1
1  5         b         6
2 10         c         7
```

We can use indexed DataFrames as well.

```
>>> left = pd.DataFrame({'left_val': ['a', 'b', 'c']}, index=[1, 5, 10])
>>> left
   left_val
1         a
5         b
10        c
```

```
>>> right = pd.DataFrame({'right_val': [1, 2, 3, 6, 7]},
...                       index=[1, 2, 3, 6, 7])
>>> right
   right_val
1          1
2          2
3          3
6          6
7          7
```

```
>>> pd.merge_asof(left, right, left_index=True, right_index=True)
   left_val right_val
1         a         1
5         b         3
10        c         7
```

Here is a real-world times-series example

```
>>> quotes
      time ticker  bid  ask
0 2016-05-25 13:30:00.023  GOOG  720.50  720.93
1 2016-05-25 13:30:00.023  MSFT  51.95  51.96
2 2016-05-25 13:30:00.030  MSFT  51.97  51.98
3 2016-05-25 13:30:00.041  MSFT  51.99  52.00
4 2016-05-25 13:30:00.048  GOOG  720.50  720.93
5 2016-05-25 13:30:00.049  AAPL  97.99  98.01
6 2016-05-25 13:30:00.072  GOOG  720.50  720.88
7 2016-05-25 13:30:00.075  MSFT  52.01  52.03
```

```
>>> trades
```

	time	ticker	price	quantity
0	2016-05-25 13:30:00.023	MSFT	51.95	75
1	2016-05-25 13:30:00.038	MSFT	51.95	155
2	2016-05-25 13:30:00.048	GOOG	720.77	100
3	2016-05-25 13:30:00.048	GOOG	720.92	100
4	2016-05-25 13:30:00.048	AAPL	98.00	100

By default we are taking the asof of the quotes

```
>>> pd.merge_asof(trades, quotes,
...               on='time',
...               by='ticker')
```

	time	ticker	price	quantity	bid	ask
0	2016-05-25 13:30:00.023	MSFT	51.95	75	51.95	51.96
1	2016-05-25 13:30:00.038	MSFT	51.95	155	51.97	51.98
2	2016-05-25 13:30:00.048	GOOG	720.77	100	720.50	720.93
3	2016-05-25 13:30:00.048	GOOG	720.92	100	720.50	720.93
4	2016-05-25 13:30:00.048	AAPL	98.00	100	NaN	NaN

We only asof within 2ms between the quote time and the trade time

```
>>> pd.merge_asof(trades, quotes,
...               on='time',
...               by='ticker',
...               tolerance=pd.Timedelta('2ms'))
```

	time	ticker	price	quantity	bid	ask
0	2016-05-25 13:30:00.023	MSFT	51.95	75	51.95	51.96
1	2016-05-25 13:30:00.038	MSFT	51.95	155	NaN	NaN
2	2016-05-25 13:30:00.048	GOOG	720.77	100	720.50	720.93
3	2016-05-25 13:30:00.048	GOOG	720.92	100	720.50	720.93
4	2016-05-25 13:30:00.048	AAPL	98.00	100	NaN	NaN

We only asof within 10ms between the quote time and the trade time and we exclude exact matches on time. However *prior* data will propagate forward

```
>>> pd.merge_asof(trades, quotes,
...               on='time',
...               by='ticker',
...               tolerance=pd.Timedelta('10ms'),
...               allow_exact_matches=False)
```

	time	ticker	price	quantity	bid	ask
0	2016-05-25 13:30:00.023	MSFT	51.95	75	NaN	NaN
1	2016-05-25 13:30:00.038	MSFT	51.95	155	51.97	51.98
2	2016-05-25 13:30:00.048	GOOG	720.77	100	NaN	NaN
3	2016-05-25 13:30:00.048	GOOG	720.92	100	NaN	NaN
4	2016-05-25 13:30:00.048	AAPL	98.00	100	NaN	NaN

### 34.2.1.10 pandas.concat

`pandas.concat` (*objs*, *axis=0*, *join='outer'*, *join\_axes=None*, *ignore\_index=False*, *keys=None*, *levels=None*, *names=None*, *verify\_integrity=False*, *sort=None*, *copy=True*)

Concatenate pandas objects along a particular axis with optional set logic along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis, which may be useful if the labels are the same (or overlapping) on the passed axis number.



**Parameters** **objs** : a sequence or mapping of Series, DataFrame, or Panel objects

If a dict is passed, the sorted keys will be used as the *keys* argument, unless it is passed, in which case the values will be selected (see below). Any None objects will be dropped silently unless they are all None in which case a ValueError will be raised

**axis** : {0/'index', 1/'columns'}, default 0

The axis to concatenate along

**join** : {'inner', 'outer'}, default 'outer'

How to handle indexes on other axis(es)

**join\_axes** : list of Index objects

Specific indexes to use for the other  $n - 1$  axes instead of performing inner/outer set logic

**ignore\_index** : boolean, default False

If True, do not use the index values along the concatenation axis. The resulting axis will be labeled 0, ...,  $n - 1$ . This is useful if you are concatenating objects where the concatenation axis does not have meaningful indexing information. Note the index values on the other axes are still respected in the join.

**keys** : sequence, default None

If multiple levels passed, should contain tuples. Construct hierarchical index using the passed keys as the outermost level

**levels** : list of sequences, default None

Specific levels (unique values) to use for constructing a MultiIndex. Otherwise they will be inferred from the keys

**names** : list, default None

Names for the levels in the resulting hierarchical index

**verify\_integrity** : boolean, default False

Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation

**sort** : boolean, default None

Sort non-concatenation axis if it is not already aligned when *join* is 'outer'. The current default of sorting is deprecated and will change to not-sorting in a future version of pandas.

Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

This has no effect when `join='inner'`, which already preserves the order of the non-concatenation axis.

New in version 0.23.0.

**copy** : boolean, default True

If False, do not copy data unnecessarily

**Returns** **concatenated** : object, type of objs

When concatenating all `Series` along the index (`axis=0`), a `Series` is returned. When `objs` contains at least one `DataFrame`, a `DataFrame` is returned. When concatenating along the columns (`axis=1`), a `DataFrame` is returned.

**See also:**

*`Series.append`, `DataFrame.append`, `DataFrame.join`, `DataFrame.merge`*

**Notes**

The `keys`, `levels`, and `names` arguments are all optional.

A walkthrough of how this method fits in with other tools for combining pandas objects can be found [here](#).

**Examples**

Combine two `Series`.

```
>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object
```

Clear the existing index and reset it in the result by setting the `ignore_index` option to `True`.

```
>>> pd.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object
```

Add a hierarchical index at the outermost level of the data with the `keys` option.

```
>>> pd.concat([s1, s2], keys=['s1', 's2',])
s1 0    a
   1    b
s2 0    c
   1    d
dtype: object
```

Label the index keys you create with the `names` option.

```
>>> pd.concat([s1, s2], keys=['s1', 's2'],
...           names=['Series name', 'Row ID'])
Series name  Row ID
s1          0      a
           1      b
s2          0      c
           1      d
dtype: object
```

Combine two `DataFrame` objects with identical columns.

```

>>> df1 = pd.DataFrame([['a', 1], ['b', 2]],
...                     columns=['letter', 'number'])
>>> df1
  letter  number
0      a        1
1      b        2
>>> df2 = pd.DataFrame([['c', 3], ['d', 4]],
...                     columns=['letter', 'number'])
>>> df2
  letter  number
0      c        3
1      d        4
>>> pd.concat([df1, df2])
  letter  number
0      a        1
1      b        2
0      c        3
1      d        4

```

Combine DataFrame objects with overlapping columns and return everything. Columns outside the intersection will be filled with NaN values.

```

>>> df3 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
...                     columns=['letter', 'number', 'animal'])
>>> df3
  letter  number animal
0      c        3   cat
1      d        4   dog
>>> pd.concat([df1, df3])
  animal letter  number
0    NaN      a        1
1    NaN      b        2
0    cat      c        3
1    dog      d        4

```

Combine DataFrame objects with overlapping columns and return only those that are shared by passing inner to the join keyword argument.

```

>>> pd.concat([df1, df3], join="inner")
  letter  number
0      a        1
1      b        2
0      c        3
1      d        4

```

Combine DataFrame objects horizontally along the x axis by passing in axis=1.

```

>>> df4 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george']],
...                     columns=['animal', 'name'])
>>> pd.concat([df1, df4], axis=1)
  letter  number animal   name
0      a        1   bird  polly
1      b        2 monkey george

```

Prevent the result from including duplicate index values with the verify\_integrity option.

```
>>> df5 = pd.DataFrame([1], index=['a'])
>>> df5
0
a 1
>>> df6 = pd.DataFrame([2], index=['a'])
>>> df6
0
a 2
>>> pd.concat([df5, df6], verify_integrity=True)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: ['a']
```

### 34.2.1.11 pandas.get\_dummies

`pandas.get_dummies` (*data*, *prefix=None*, *prefix\_sep='\_'*, *dummy\_na=False*, *columns=None*,  
*sparse=False*, *drop\_first=False*, *dtype=None*)  
Convert categorical variable into dummy/indicator variables

#### Parameters

**data** [array-like, Series, or DataFrame]

**prefix** : string, list of strings, or dict of strings, default None

String to append DataFrame column names. Pass a list with length equal to the number of columns when calling `get_dummies` on a DataFrame. Alternatively, *prefix* can be a dictionary mapping column names to prefixes.

**prefix\_sep** : string, default '\_'

If appending prefix, separator/delimiter to use. Or pass a list or dictionary as with *prefix*.

**dummy\_na** : bool, default False

Add a column to indicate NaNs, if False NaNs are ignored.

**columns** : list-like, default None

Column names in the DataFrame to be encoded. If *columns* is None then all the columns with *object* or *category* dtype will be converted.

**sparse** : bool, default False

Whether the dummy columns should be sparse or not. Returns `SparseDataFrame` if *data* is a Series or if all columns are included. Otherwise returns a DataFrame with some `SparseBlocks`.

**drop\_first** : bool, default False

Whether to get k-1 dummies out of k categorical levels by removing the first level.

New in version 0.18.0.

**dtype** : dtype, default np.uint8

Data type for new columns. Only a single dtype is allowed.

New in version 0.23.0.

#### Returns

**dummies** [DataFrame or SparseDataFrame]

**See also:***Series.str.get\_dummies***Examples**

```
>>> import pandas as pd
>>> s = pd.Series(list('abca'))
```

```
>>> pd.get_dummies(s)
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
```

```
>>> s1 = ['a', 'b', np.nan]
```

```
>>> pd.get_dummies(s1)
   a  b
0  1  0
1  0  1
2  0  0
```

```
>>> pd.get_dummies(s1, dummy_na=True)
   a  b  NaN
0  1  0    0
1  0  1    0
2  0  0    1
```

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['b', 'a', 'c'],
...                     'C': [1, 2, 3]})
```

```
>>> pd.get_dummies(df, prefix=['col1', 'col2'])
   C  col1_a  col1_b  col2_a  col2_b  col2_c
0  1         1         0         0         1         0
1  2         0         1         1         0         0
2  3         1         0         0         0         1
```

```
>>> pd.get_dummies(pd.Series(list('abcaa')))
   a  b  c
0  1  0  0
1  0  1  0
2  0  0  1
3  1  0  0
4  1  0  0
```

```
>>> pd.get_dummies(pd.Series(list('abcaa')), drop_first=True)
   b  c
0  0  0
1  1  0
2  0  1
3  0  0
4  0  0
```

```
>>> pd.get_dummies(pd.Series(list('abc')), dtype=float)
   a    b    c
0  1.0  0.0  0.0
1  0.0  1.0  0.0
2  0.0  0.0  1.0
```

### 34.2.1.12 pandas.factorize

pandas.**factorize** (*values*, *sort=False*, *order=None*, *na\_sentinel=-1*, *size\_hint=None*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

**Parameters** *values* : sequence

A 1-D sequence. Sequences that aren't pandas objects are coerced to ndarrays before factorization.

**sort** : bool, default False

Sort *uniques* and shuffle *labels* to maintain the relationship.

**order**

Deprecated since version 0.23.0: This parameter has no effect and is deprecated.

**na\_sentinel** : int, default -1

Value to mark “not found”.

**size\_hint** : int, optional

Hint to the hashtable sizer.

**Returns** *labels* : ndarray

An integer ndarray that's an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

**uniques** : ndarray, Index, or Categorical

The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

---

**Note:** Even if there's a missing value in *values*, *uniques* will *not* contain an entry for it.

---

**See also:**

`pandas.cut` Discretize continuous-valued array.

`pandas.unique` Find the unique valuse in an array.

## Examples

These examples all show `factorize` as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *labels* with *na\_sentinel* (-1 by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that 'b' is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

### 34.2.1.13 pandas.unique

`pandas.unique(values)`

Hash table-based unique. Uniques are returned in order of appearance. This does NOT sort.

Significantly faster than `numpy.unique`. Includes NA values.

**Parameters****values** [1d array-like]**Returns** unique values.

- If the input is an Index, the return is an Index
- If the input is a Categorical dtype, the return is a Categorical
- If the input is a Series/ndarray, the return will be an ndarray

**See also:***pandas.Index.unique, pandas.Series.unique***Examples**

```
>>> pd.unique(pd.Series([2, 1, 3, 3]))
array([2, 1, 3])
```

```
>>> pd.unique(pd.Series([2] + [1] * 5))
array([2, 1])
```

```
>>> pd.unique(Series([pd.Timestamp('20160101'),
...                   pd.Timestamp('20160101')]))
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.unique(pd.Series([pd.Timestamp('20160101', tz='US/Eastern'),
...                   pd.Timestamp('20160101', tz='US/Eastern')]))
array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
      dtype=object)
```

```
>>> pd.unique(pd.Index([pd.Timestamp('20160101', tz='US/Eastern'),
...                   pd.Timestamp('20160101', tz='US/Eastern')]))
DatetimeIndex(['2016-01-01 00:00:00-05:00'],
...          dtype='datetime64[ns, US/Eastern]', freq=None)
```

```
>>> pd.unique(list('baabc'))
array(['b', 'a', 'c'], dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.unique(Series(pd.Categorical(list('baabc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

```
>>> pd.unique(Series(pd.Categorical(list('baabc'),
...                               categories=list('abc'))))
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.unique(Series(pd.Categorical(list('baabc'),
...                               categories=list('abc')),
```

(continues on next page)



(continued from previous page)

```
...                                ordered=True))
[b, a, c]
Categories (3, object): [a < b < c]
```

An array of tuples

```
>>> pd.unique([('a', 'b'), ('b', 'a'), ('a', 'c'), ('b', 'a')])
array([('a', 'b'), ('b', 'a'), ('a', 'c')], dtype=object)
```

### 34.2.1.14 pandas.wide\_to\_long

`pandas.wide_to_long(df, stubnames, i, j, sep=" ", suffix='\d+')`

Wide panel to long format. Less flexible but more user-friendly than melt.

With stubnames ['A', 'B'], this function expects to find one or more group of columns with format Asuffix1, Asuffix2,..., Bsufffix1, Bsufffix2,... You specify what you want to call this suffix in the resulting long format with *j* (for example *j*='year')

Each row of these wide variables are assumed to be uniquely identified by *i* (can be a single column name or a list of column names)

All remaining variables in the data frame are left intact.

**Parameters** **df** : DataFrame

The wide-format DataFrame

**stubnames** : str or list-like

The stub name(s). The wide format variables are assumed to start with the stub names.

**i** : str or list-like

Column(s) to use as id variable(s)

**j** : str

The name of the subobservation variable. What you wish to name your suffix in the long format.

**sep** : str, default ""

A character indicating the separation of the variable names in the wide format, to be stripped from the names in the long format. For example, if your column names are A-suffix1, A-suffix2, you can strip the hyphen by specifying *sep*='-'

New in version 0.20.0.

**suffix** : str, default '\d+'

A regular expression capturing the wanted suffixes. '\d+' captures numeric suffixes. Suffixes with no numbers could be specified with the negated character class '\D+'. You can also further disambiguate suffixes, for example, if your wide variables are of the form Aone, Btwo,..., and you have an unrelated column Arating, you can ignore the last one by specifying *suffix*='(!?one|two)'

New in version 0.20.0.

Changed in version 0.23.0: When all suffixes are numeric, they are cast to int64/float64.

**Returns** DataFrame

A DataFrame that contains each stub name as a variable, with new index (i, j)

## Notes

All extra variables are left untouched. This simply uses *pandas.melt* under the hood, but is hard-coded to “do the right thing” in a typical case.

## Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> np.random.seed(123)
>>> df = pd.DataFrame({"A1970" : {0 : "a", 1 : "b", 2 : "c"},
...                     "A1980" : {0 : "d", 1 : "e", 2 : "f"},
...                     "B1970" : {0 : 2.5, 1 : 1.2, 2 : .7},
...                     "B1980" : {0 : 3.2, 1 : 1.3, 2 : .1},
...                     "X"      : dict(zip(range(3), np.random.randn(3)))
...                     })
>>> df["id"] = df.index
>>> df
   A1970 A1980  B1970  B1980      X  id
0      a      d    2.5    3.2 -1.085631  0
1      b      e    1.2    1.3  0.997345  1
2      c      f    0.7    0.1  0.282978  2
>>> pd.wide_to_long(df, ["A", "B"], i="id", j="year")
...
      X  A    B
id year
0  1970 -1.085631  a  2.5
1  1970  0.997345  b  1.2
2  1970  0.282978  c  0.7
0  1980 -1.085631  d  3.2
1  1980  0.997345  e  1.3
2  1980  0.282978  f  0.1
```

### With multiple id columns

```
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht1': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
...     'ht2': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
   birth  famid  ht1  ht2
0      1      1  2.8  3.4
1      2      1  2.9  3.8
2      3      1  2.2  2.9
3      1      2  2.0  3.2
4      2      2  1.8  2.8
5      3      2  1.9  2.4
6      1      3  2.2  3.3
7      2      3  2.3  3.4
8      3      3  2.1  2.9
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age')
```

(continues on next page)

(continued from previous page)

```
>>> l
...
      ht
famid birth age
1      1      1  2.8
      2      2  3.4
      2      1  2.9
      2      2  3.8
      3      1  2.2
      3      2  2.9
2      1      1  2.0
      2      2  3.2
      2      1  1.8
      2      2  2.8
      3      1  1.9
      3      2  2.4
3      1      1  2.2
      2      2  3.3
      2      1  2.3
      2      2  3.4
      3      1  2.1
      3      2  2.9
```

Going from long back to wide just takes some creative use of *unstack*

```
>>> w = l.unstack()
>>> w.columns = w.columns.map('{0[0]}{0[1]}'.format)
>>> w.reset_index()
   famid  birth  ht1  ht2
0      1      1  2.8  3.4
1      1      2  2.9  3.8
2      1      3  2.2  2.9
3      2      1  2.0  3.2
4      2      2  1.8  2.8
5      2      3  1.9  2.4
6      3      1  2.2  3.3
7      3      2  2.3  3.4
8      3      3  2.1  2.9
```

Less wieldy column names are also handled

```
>>> np.random.seed(0)
>>> df = pd.DataFrame({'A(quarterly)-2010': np.random.rand(3),
...                    'A(quarterly)-2011': np.random.rand(3),
...                    'B(quarterly)-2010': np.random.rand(3),
...                    'B(quarterly)-2011': np.random.rand(3),
...                    'X' : np.random.randint(3, size=3)})
>>> df['id'] = df.index
>>> df
   A(quarterly)-2010  A(quarterly)-2011  B(quarterly)-2010  ...
0          0.548814          0.544883          0.437587  ...
1          0.715189          0.423655          0.891773  ...
2          0.602763          0.645894          0.963663  ...
   X  id
0  0   0
1  1   1
2  1   2
```

```
>>> pd.wide_to_long(df, ['A(quarterly)', 'B(quarterly)'], i='id',
...                  j='year', sep='-')
...
      X  A(quarterly)  B(quarterly)
id year
0  2010    0      0.548814    0.437587
1  2010    1      0.715189    0.891773
2  2010    1      0.602763    0.963663
0  2011    0      0.544883    0.383442
1  2011    1      0.423655    0.791725
2  2011    1      0.645894    0.528895
```

If we have many columns, we could also use a regex to find our stubnames and pass that list on to `wide_to_long`

```
>>> stubnames = sorted(
...     set([match[0] for match in df.columns.str.findall(
...         r'[A-B]\(.*\)').values if match != [] ]))
... )
>>> list(stubnames)
['A(quarterly)', 'B(quarterly)']
```

All of the above examples have integers as suffixes. It is possible to have non-integers as suffixes.

```
>>> df = pd.DataFrame({
...     'famid': [1, 1, 1, 2, 2, 2, 3, 3, 3],
...     'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
...     'ht_one': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
...     'ht_two': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]
... })
>>> df
   birth  famid  ht_one  ht_two
0      1      1    2.8     3.4
1      2      1    2.9     3.8
2      3      1    2.2     2.9
3      1      2    2.0     3.2
4      2      2    1.8     2.8
5      3      2    1.9     2.4
6      1      3    2.2     3.3
7      2      3    2.3     3.4
8      3      3    2.1     2.9
```

```
>>> l = pd.wide_to_long(df, stubnames='ht', i=['famid', 'birth'], j='age',
...                      sep='_', suffix='\w')
>>> l
...
      ht
famid birth age
1      1    one 2.8
      two 3.4
      2    one 2.9
      two 3.8
      3    one 2.2
      two 2.9
2      1    one 2.0
      two 3.2
      2    one 1.8
      two 2.8
```

(continues on next page)

(continued from previous page)

	3	one	1.9
		two	2.4
3	1	one	2.2
		two	3.3
	2	one	2.3
		two	3.4
	3	one	2.1
		two	2.9

### 34.2.2 Top-level missing data

<code>isna(obj)</code>	Detect missing values for an array-like object.
<code>isnull(obj)</code>	Detect missing values for an array-like object.
<code>notna(obj)</code>	Detect non-missing values for an array-like object.
<code>notnull(obj)</code>	Detect non-missing values for an array-like object.

#### 34.2.2.1 pandas.isna

`pandas.isna(obj)`

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

**Parameters** `obj` : scalar or array-like

Object to check for null or missing values.

**Returns** bool or array-like of bool

For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

**See also:**

`notna` boolean inverse of `pandas.isna`.

`Series.isna` Detect missing values in a Series.

`DataFrame.isna` Detect missing values in a DataFrame.

`Index.isna` Detect missing values in an Index.

#### Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame(['ant', 'bee', 'cat'], ['dog', None, 'fly'])
>>> df
   0    1    2
0  ant  bee  cat
1  dog None fly
>>> pd.isna(df)
   0    1    2
0 False False False
1 False  True False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

### 34.2.2.2 pandas.isnull

pandas.**isnull** (*obj*)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

**Parameters** *obj* : scalar or array-like

Object to check for null or missing values.

**Returns** bool or array-like of bool

For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

**See also:**

[\*notna\*](#) boolean inverse of pandas.isna.

[\*Series.isna\*](#) Detect missing values in a Series.

[\*DataFrame.isna\*](#) Detect missing values in a DataFrame.

**Index.isna** Detect missing values in an Index.

## Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame(['ant', 'bee', 'cat'], ['dog', None, 'fly'])
>>> df
   0    1    2
0  ant  bee  cat
1  dog None fly
>>> pd.isna(df)
   0    1    2
0 False False False
1 False  True False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

### 34.2.2.3 pandas.notna

pandas.**notna** (*obj*)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

**Parameters** `obj` : array-like or object value

Object to check for *not* null or *non*-missing values.

**Returns** bool or array-like of bool

For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

**See also:**

[`isna`](#) boolean inverse of `pandas.notna`.

[`Series.notna`](#) Detect valid values in a Series.

[`DataFrame.notna`](#) Detect valid values in a DataFrame.

[`Index.notna`](#) Detect valid values in an Index.

## Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame(['ant', 'bee', 'cat'], ['dog', None, 'fly'])
>>> df
   0    1    2
0 ant  bee cat
```

(continues on next page)



(continued from previous page)

```

1  dog  None  fly
>>> pd.isna(df)
      0      1      2
0  True   True   True
1  True  False   True

```

```

>>> pd.isna(df[1])
0      True
1     False
Name: 1, dtype: bool

```

### 34.2.2.4 pandas.isnull

`pandas.isnull(obj)`

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

**Parameters** `obj` : array-like or object value

Object to check for *not* null or *non*-missing values.

**Returns** `bool` or `array-like of bool`

For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

**See also:**

[`isna`](#) boolean inverse of `pandas.isna`.

[`Series.isna`](#) Detect valid values in a Series.

[`DataFrame.isna`](#) Detect valid values in a DataFrame.

[`Index.isna`](#) Detect valid values in an Index.

### Examples

Scalar arguments (including strings) result in a scalar boolean.

```

>>> pd.isna('dog')
True

```

```

>>> pd.isna(np.nan)
False

```

ndarrays result in an ndarray of booleans.

```

>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)

```

(continues on next page)

(continued from previous page)

```
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame(['ant', 'bee', 'cat'], ['dog', None, 'fly'])
>>> df
   0    1    2
0  ant  bee  cat
1  dog None fly
>>> pd.notna(df)
   0    1    2
0  True  True  True
1  True False  True
```

```
>>> pd.notna(df[1])
0    True
1   False
Name: 1, dtype: bool
```

### 34.2.3 Top-level conversions

---

`to_numeric(arg[, errors, downcast])`

---

Convert argument to a numeric type.

#### 34.2.3.1 pandas.to\_numeric

`pandas.to_numeric(arg, errors='raise', downcast=None)`

Convert argument to a numeric type.

##### Parameters

**arg** [list, tuple, 1-d array, or Series]

**errors** : { 'ignore', 'raise', 'coerce' }, default 'raise'

- If 'raise', then invalid parsing will raise an exception
- If 'coerce', then invalid parsing will be set as NaN
- If 'ignore', then invalid parsing will return the input

**downcast** : { 'integer', 'signed', 'unsigned', 'float' }, default None

If not None, and if the data has been successfully cast to a numerical dtype (or if the data was numeric to begin with), downcast that resulting data to the smallest numerical dtype possible according to the following rules:

- ‘integer’ or ‘signed’: smallest signed int dtype (min.: np.int8)
- ‘unsigned’: smallest unsigned int dtype (min.: np.uint8)
- ‘float’: smallest float dtype (min.: np.float32)

As this behaviour is separate from the core conversion to numeric values, any errors raised during the downcasting will be surfaced regardless of the value of the ‘errors’ input.

In addition, downcasting will only occur if the size of the resulting data’s dtype is strictly larger than the dtype it is to be cast to, so if none of the dtypes checked satisfy that specification, no downcasting will be performed on the data.

New in version 0.19.0.

**Returns** `ret` : numeric if parsing succeeded.

Return type depends on input. Series if Series, otherwise ndarray

**See also:**

`pandas.DataFrame.astype` Cast argument to a specified dtype.

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`numpy.ndarray.astype` Cast a numpy array to a specified type.

## Examples

Take separate series and convert to numeric, coercing when told to

```
>>> import pandas as pd
>>> s = pd.Series(['1.0', '2', -3])
>>> pd.to_numeric(s)
0    1.0
1    2.0
2   -3.0
dtype: float64
>>> pd.to_numeric(s, downcast='float')
0    1.0
1    2.0
2   -3.0
dtype: float32
>>> pd.to_numeric(s, downcast='signed')
0     1
1     2
2    -3
dtype: int8
>>> s = pd.Series(['apple', '1.0', '2', -3])
>>> pd.to_numeric(s, errors='ignore')
0    apple
1     1.0
2      2
3     -3
dtype: object
>>> pd.to_numeric(s, errors='coerce')
0     NaN
```

(continues on next page)

(continued from previous page)

```

1      1.0
2      2.0
3     -3.0
dtype: float64

```

### 34.2.4 Top-level dealing with datetimelike

<code>to_datetime(arg[, errors, dayfirst, ...])</code>	Convert argument to datetime.
<code>to_timedelta(arg[, unit, box, errors])</code>	Convert argument to timedelta
<code>date_range([start, end, periods, freq, tz, ...])</code>	Return a fixed frequency DatetimeIndex.
<code>bdate_range([start, end, periods, freq, tz, ...])</code>	Return a fixed frequency DatetimeIndex, with business day as the default frequency
<code>period_range([start, end, periods, freq, name])</code>	Return a fixed frequency PeriodIndex, with day (calendar) as the default frequency
<code>timedelta_range([start, end, periods, freq, ...])</code>	Return a fixed frequency TimedeltaIndex, with day as the default frequency
<code>infer_freq(index[, warn])</code>	Infer the most likely frequency given the input index.

#### 34.2.4.1 pandas.to\_datetime

`pandas.to_datetime` (*arg*, *errors*='raise', *dayfirst*=False, *yearfirst*=False, *utc*=None, *box*=True, *format*=None, *exact*=True, *unit*=None, *infer\_datetime\_format*=False, *origin*='unix', *cache*=False)

Convert argument to datetime.

**Parameters** *arg* : integer, float, string, datetime, list, tuple, 1-d array, Series

New in version 0.18.1: or DataFrame/dict-like

**errors** : { 'ignore', 'raise', 'coerce' }, default 'raise'

- If 'raise', then invalid parsing will raise an exception
- If 'coerce', then invalid parsing will be set as NaT
- If 'ignore', then invalid parsing will return the input

**dayfirst** : boolean, default False

Specify a date parse order if *arg* is str or its list-likes. If True, parses dates with the day first, eg 10/11/12 is parsed as 2012-11-10. Warning: dayfirst=True is not strict, but will prefer to parse with day first (this is a known bug, based on dateutil behavior).

**yearfirst** : boolean, default False

Specify a date parse order if *arg* is str or its list-likes.

- If True parses dates with the year first, eg 10/11/12 is parsed as 2010-11-12.
- If both dayfirst and yearfirst are True, yearfirst is preceded (same as dateutil).

Warning: yearfirst=True is not strict, but will prefer to parse with year first (this is a known bug, based on dateutil behavior).

New in version 0.16.1.

**utc** : boolean, default None

Return UTC DatetimeIndex if True (converting any tz-aware datetime.datetime objects as well).

**box** : boolean, default True

- If True returns a DatetimeIndex
- If False returns ndarray of values.

**format** : string, default None

strftime to parse time, eg “%d/%m/%Y”, note that “%f” will parse all the way up to nanoseconds.

**exact** : boolean, True by default

- If True, require an exact format match.
- If False, allow the format to match anywhere in the target string.

**unit** : string, default ‘ns’

unit of the arg (D,s,ms,us,ns) denote the unit, which is an integer or float number. This will be based off the origin. Example, with unit=‘ms’ and origin=‘unix’ (the default), this would calculate the number of milliseconds to the unix epoch start.

**infer\_datetime\_format** : boolean, default False

If True and no *format* is given, attempt to infer the format of the datetime strings, and if it can be inferred, switch to a faster method of parsing them. In some cases this can increase the parsing speed by ~5-10x.

**origin** : scalar, default is ‘unix’

Define the reference date. The numeric values would be parsed as number of units (defined by *unit*) since this reference date.

- If ‘unix’ (or POSIX) time; origin is set to 1970-01-01.
- If ‘julian’, unit must be ‘D’, and origin is set to beginning of Julian Calendar. Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.
- If Timestamp convertible, origin is set to Timestamp identified by origin.

New in version 0.20.0.

**cache** : boolean, default False

If True, use a cache of unique, converted dates to apply the datetime conversion. May produce significant speed-up when parsing duplicate date strings, especially ones with timezone offsets.

New in version 0.23.0.

**Returns** **ret** : datetime if parsing succeeded.

Return type depends on input:

- list-like: DatetimeIndex
- Series: Series of datetime64 dtype
- scalar: Timestamp

In case when it is not possible to return designated types (e.g. when any element of input is before Timestamp.min or after Timestamp.max) return will have datetime.datetime type (or corresponding array/Series).

See also:

`pandas.DataFrame.astype` Cast argument to a specified dtype.

`pandas.to_timedelta` Convert argument to timedelta.

## Examples

Assembling a datetime from multiple columns of a DataFrame. The keys can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same

```
>>> df = pd.DataFrame({'year': [2015, 2016],
                        'month': [2, 3],
                        'day': [4, 5]})
>>> pd.to_datetime(df)
0    2015-02-04
1    2016-03-05
dtype: datetime64[ns]
```

If a date does not meet the [timestamp limitations](#), passing `errors='ignore'` will return the original input instead of raising any exception.

Passing `errors='coerce'` will force an out-of-bounds date to NaT, in addition to forcing non-dates (or non-parseable dates) to NaT.

```
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='ignore')
datetime.datetime(1300, 1, 1, 0, 0)
>>> pd.to_datetime('13000101', format='%Y%m%d', errors='coerce')
NaT
```

Passing `infer_datetime_format=True` can often-times speedup a parsing if its not an ISO8601 format exactly, but in a regular format.

```
>>> s = pd.Series(['3/11/2000', '3/12/2000', '3/13/2000']*1000)
```

```
>>> s.head()
0    3/11/2000
1    3/12/2000
2    3/13/2000
3    3/11/2000
4    3/12/2000
dtype: object
```

```
>>> %timeit pd.to_datetime(s,infer_datetime_format=True)
100 loops, best of 3: 10.4 ms per loop
```

```
>>> %timeit pd.to_datetime(s,infer_datetime_format=False)
1 loop, best of 3: 471 ms per loop
```

Using a unix epoch time

```
>>> pd.to_datetime(1490195805, unit='s')
Timestamp('2017-03-22 15:16:45')
>>> pd.to_datetime(1490195805433502912, unit='ns')
Timestamp('2017-03-22 15:16:45.433502912')
```

**Warning:** For float arg, precision rounding might happen. To prevent unexpected behavior use a fixed-width exact type.

Using a non-unix epoch origin

```
>>> pd.to_datetime([1, 2, 3], unit='D',
                    origin=pd.Timestamp('1960-01-01'))
0    1960-01-02
1    1960-01-03
2    1960-01-04
```

#### 34.2.4.2 pandas.to\_timedelta

`pandas.to_timedelta(arg, unit='ns', box=True, errors='raise')`

Convert argument to timedelta

##### Parameters

**arg** [string, timedelta, list, tuple, 1-d array, or Series]

**unit** : unit of the arg (D,h,m,s,ms,us,ns) denote the unit, which is an integer/float number

**box** : boolean, default True

- If True returns a Timedelta/TimedeltaIndex of the results
- if False returns a np.timedelta64 or ndarray of values of dtype timedelta64[ns]

**errors** : {'ignore', 'raise', 'coerce'}, default 'raise'

- If 'raise', then invalid parsing will raise an exception
- If 'coerce', then invalid parsing will be set as NaT
- If 'ignore', then invalid parsing will return the input

##### Returns

**ret** [timedelta64/arrays of timedelta64 if parsing succeeded]

See also:

[`pandas.DataFrame.astype`](#) Cast argument to a specified dtype.

[`pandas.to\_datetime`](#) Convert argument to datetime.

#### Examples

Parsing a single string to a Timedelta:

```
>>> pd.to_timedelta('1 days 06:05:01.00003')
Timedelta('1 days 06:05:01.000030')
>>> pd.to_timedelta('15.5us')
Timedelta('0 days 00:00:00.000015')
```

Parsing a list or array of strings:

```
>>> pd.to_timedelta(['1 days 06:05:01.00003', '15.5us', 'nan'])
TimedeltaIndex(['1 days 06:05:01.000030', '0 days 00:00:00.000015', NaT],
               dtype='timedelta64[ns]', freq=None)
```

Converting numbers by specifying the *unit* keyword argument:

```
>>> pd.to_timedelta(np.arange(5), unit='s')
TimedeltaIndex(['00:00:00', '00:00:01', '00:00:02',
               '00:00:03', '00:00:04'],
               dtype='timedelta64[ns]', freq=None)
>>> pd.to_timedelta(np.arange(5), unit='d')
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

### 34.2.4.3 pandas.date\_range

`pandas.date_range` (*start=None, end=None, periods=None, freq=None, tz=None, normalize=False, name=None, closed=None, \*\*kwargs*)

Return a fixed frequency DatetimeIndex.

**Parameters** **start** : str or datetime-like, optional

Left bound for generating dates.

**end** : str or datetime-like, optional

Right bound for generating dates.

**periods** : integer, optional

Number of periods to generate.

**freq** : str or DateOffset, default 'D' (calendar daily)

Frequency strings can have multiples, e.g. '5H'. See [here](#) for a list of frequency aliases.

**tz** : str or tzinfo, optional

Time zone name for returning localized DatetimeIndex, for example 'Asia/Hong\_Kong'. By default, the resulting DatetimeIndex is timezone-naive.

**normalize** : bool, default False

Normalize start/end dates to midnight before generating date range.

**name** : str, default None

Name of the resulting DatetimeIndex.

**closed** : {None, 'left', 'right'}, optional

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None, the default).

**\*\*kwargs**

For compatibility. Has no effect on the result.

**Returns**

**rng** [DatetimeIndex]

See also:



**`pandas.DatetimeIndex`** An immutable container for datetimes.

**`pandas.timedelta_range`** Return a fixed frequency TimedeltaIndex.

**`pandas.period_range`** Return a fixed frequency PeriodIndex.

**`pandas.interval_range`** Return a fixed frequency IntervalIndex.

## Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `DatetimeIndex` will have periods linearly spaced elements between `start` and `end` (closed on both sides).

To learn more about the frequency strings, please see [this link](#).

## Examples

### Specifying the values

The next four examples generate the same `DatetimeIndex`, but vary the combination of `start`, `end` and `periods`.

Specify `start` and `end`, with the default daily frequency.

```
>>> pd.date_range(start='1/1/2018', end='1/08/2018')
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify `start` and `periods`, the number of periods (days).

```
>>> pd.date_range(start='1/1/2018', periods=8)
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06', '2018-01-07', '2018-01-08'],
              dtype='datetime64[ns]', freq='D')
```

Specify `end` and `periods`, the number of periods (days).

```
>>> pd.date_range(end='1/1/2018', periods=8)
DatetimeIndex(['2017-12-25', '2017-12-26', '2017-12-27', '2017-12-28',
               '2017-12-29', '2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

Specify `start`, `end`, and `periods`; the frequency is generated automatically (linearly spaced).

```
>>> pd.date_range(start='2018-04-24', end='2018-04-27', periods=3)
DatetimeIndex(['2018-04-24 00:00:00', '2018-04-25 12:00:00',
               '2018-04-27 00:00:00'], freq=None)
```

### Other Parameters

Changed the `freq` (frequency) to 'M' (month end frequency).

```
>>> pd.date_range(start='1/1/2018', periods=5, freq='M')
DatetimeIndex(['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30',
               '2018-05-31'],
              dtype='datetime64[ns]', freq='M')
```

Multiples are allowed

```
>>> pd.date_range(start='1/1/2018', periods=5, freq='3M')
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
              '2019-01-31'],
              dtype='datetime64[ns]', freq='3M')
```

*freq* can also be specified as an Offset object.

```
>>> pd.date_range(start='1/1/2018', periods=5, freq=pd.offsets.MonthEnd(3))
DatetimeIndex(['2018-01-31', '2018-04-30', '2018-07-31', '2018-10-31',
              '2019-01-31'],
              dtype='datetime64[ns]', freq='3M')
```

Specify *tz* to set the timezone.

```
>>> pd.date_range(start='1/1/2018', periods=5, tz='Asia/Tokyo')
DatetimeIndex(['2018-01-01 00:00:00+09:00', '2018-01-02 00:00:00+09:00',
              '2018-01-03 00:00:00+09:00', '2018-01-04 00:00:00+09:00',
              '2018-01-05 00:00:00+09:00'],
              dtype='datetime64[ns, Asia/Tokyo]', freq='D')
```

*closed* controls whether to include *start* and *end* that are on the boundary. The default includes boundary points on either end.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed=None)
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

Use *closed='left'* to exclude *end* if it falls on the boundary.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='left')
DatetimeIndex(['2017-01-01', '2017-01-02', '2017-01-03'],
              dtype='datetime64[ns]', freq='D')
```

Use *closed='right'* to exclude *start* if it falls on the boundary.

```
>>> pd.date_range(start='2017-01-01', end='2017-01-04', closed='right')
DatetimeIndex(['2017-01-02', '2017-01-03', '2017-01-04'],
              dtype='datetime64[ns]', freq='D')
```

#### 34.2.4.4 pandas.bdate\_range

`pandas.bdate_range` (*start=None, end=None, periods=None, freq='B', tz=None, normalize=True, name=None, weekmask=None, holidays=None, closed=None, \*\*kwargs*)

Return a fixed frequency DatetimeIndex, with business day as the default frequency

**Parameters** *start* : string or datetime-like, default None

Left bound for generating dates

*end* : string or datetime-like, default None

Right bound for generating dates

*periods* : integer, default None

Number of periods to generate

*freq* : string or DateOffset, default 'B' (business daily)

Frequency strings can have multiples, e.g. '5H'

**tz** : string or None

Time zone name for returning localized DatetimeIndex, for example Asia/Beijing

**normalize** : bool, default False

Normalize start/end dates to midnight before generating date range

**name** : string, default None

Name of the resulting DatetimeIndex

**weekmask** : string or None, default None

Weekmask of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed. The default value None is equivalent to 'Mon Tue Wed Thu Fri'

New in version 0.21.0.

**holidays** : list-like or None, default None

Dates to exclude from the set of valid business days, passed to `numpy.busdaycalendar`, only used when custom frequency strings are passed

New in version 0.21.0.

**closed** : string, default None

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

#### Returns

**rng** [DatetimeIndex]

#### Notes

Of the four parameters: `start`, `end`, `periods`, and `freq`, exactly three must be specified. Specifying `freq` is a requirement for `bdate_range`. Use `date_range` if specifying `freq` is not desired.

To learn more about the frequency strings, please see [this link](#).

#### 34.2.4.5 pandas.period\_range

`pandas.period_range` (*start=None, end=None, periods=None, freq='D', name=None*)

Return a fixed frequency PeriodIndex, with day (calendar) as the default frequency

**Parameters** **start** : string or period-like, default None

Left bound for generating periods

**end** : string or period-like, default None

Right bound for generating periods

**periods** : integer, default None

Number of periods to generate

**freq** : string or DateOffset, default 'D' (calendar daily)

Frequency alias

**name** : string, default None

Name of the resulting PeriodIndex

#### Returns

**prng** [PeriodIndex]

#### Notes

Of the three parameters: start, end, and periods, exactly two must be specified.

To learn more about the frequency strings, please see [this link](#).

#### Examples

```
>>> pd.period_range(start='2017-01-01', end='2018-01-01', freq='M')
PeriodIndex(['2017-01', '2017-02', '2017-03', '2017-04', '2017-05',
            '2017-06', '2017-06', '2017-07', '2017-08', '2017-09',
            '2017-10', '2017-11', '2017-12', '2018-01'],
            dtype='period[M]', freq='M')
```

If start or end are Period objects, they will be used as anchor endpoints for a PeriodIndex with frequency matching that of the period\_range constructor.

```
>>> pd.period_range(start=pd.Period('2017Q1', freq='Q'),
...                 end=pd.Period('2017Q2', freq='Q'), freq='M')
PeriodIndex(['2017-03', '2017-04', '2017-05', '2017-06'],
            dtype='period[M]', freq='M')
```

#### 34.2.4.6 pandas.timedelta\_range

`pandas.timedelta_range` (*start=None, end=None, periods=None, freq=None, name=None, closed=None*)

Return a fixed frequency TimedeltaIndex, with day as the default frequency

**Parameters** **start** : string or timedelta-like, default None

Left bound for generating timedeltas

**end** : string or timedelta-like, default None

Right bound for generating timedeltas

**periods** : integer, default None

Number of periods to generate

**freq** : string or DateOffset, default 'D' (calendar daily)

Frequency strings can have multiples, e.g. '5H'

**name** : string, default None

Name of the resulting TimedeltaIndex

**closed** : string, default None

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

**Returns****rng** [TimedeltaIndex]**Notes**

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `TimedeltaIndex` will have periods linearly spaced elements between `start` and `end` (closed on both sides).

To learn more about the frequency strings, please see [this link](#).

**Examples**

```
>>> pd.timedelta_range(start='1 day', periods=4)
TimedeltaIndex(['1 days', '2 days', '3 days', '4 days'],
                dtype='timedelta64[ns]', freq='D')
```

The `closed` parameter specifies which endpoint is included. The default behavior is to include both endpoints.

```
>>> pd.timedelta_range(start='1 day', periods=4, closed='right')
TimedeltaIndex(['2 days', '3 days', '4 days'],
                dtype='timedelta64[ns]', freq='D')
```

The `freq` parameter specifies the frequency of the `TimedeltaIndex`. Only fixed frequencies can be passed, non-fixed frequencies such as 'M' (month end) will raise.

```
>>> pd.timedelta_range(start='1 day', end='2 days', freq='6H')
TimedeltaIndex(['1 days 00:00:00', '1 days 06:00:00', '1 days 12:00:00',
                '1 days 18:00:00', '2 days 00:00:00'],
                dtype='timedelta64[ns]', freq='6H')
```

Specify `start`, `end`, and `periods`; the frequency is generated automatically (linearly spaced).

```
>>> pd.timedelta_range(start='1 day', end='5 days', periods=4)
TimedeltaIndex(['1 days 00:00:00', '2 days 08:00:00', '3 days 16:00:00',
                '5 days 00:00:00'],
                dtype='timedelta64[ns]', freq=None)
```

**34.2.4.7 pandas.infer\_freq**

`pandas.infer_freq(index, warn=True)`

Infer the most likely frequency given the input index. If the frequency is uncertain, a warning will be printed.

**Parameters** `index` : DatetimeIndex or TimedeltaIndex

if passed a Series will use the values of the series (NOT THE INDEX)

**warn** [boolean, default True]

**Returns** `freq` : string or None

None if no discernible frequency  
 TypeError if the index is not datetime-like  
 ValueError if there are less than three values.

## 34.2.5 Top-level dealing with intervals

---

<code>interval_range([start, end, periods, freq, ...])</code>	Return a fixed frequency IntervalIndex
---	--

---

### 34.2.5.1 pandas.interval\_range

`pandas.interval_range` (*start=None, end=None, periods=None, freq=None, name=None, closed='right'*)

Return a fixed frequency IntervalIndex

**Parameters** **start** : numeric or datetime-like, default None

Left bound for generating intervals

**end** : numeric or datetime-like, default None

Right bound for generating intervals

**periods** : integer, default None

Number of periods to generate

**freq** : numeric, string, or DateOffset, default None

The length of each interval. Must be consistent with the type of start and end, e.g. 2 for numeric, or '5H' for datetime-like. Default is 1 for numeric and 'D' (calendar daily) for datetime-like.

**name** : string, default None

Name of the resulting IntervalIndex

**closed** : { 'left', 'right', 'both', 'neither' }, default 'right'

Whether the intervals are closed on the left-side, right-side, both or neither.

**Returns**

**rng** [IntervalIndex]

**See also:**

**IntervalIndex** an Index of intervals that are all closed on the same side.

### Notes

Of the four parameters `start`, `end`, `periods`, and `freq`, exactly three must be specified. If `freq` is omitted, the resulting `IntervalIndex` will have `periods` linearly spaced elements between `start` and `end`, inclusively.

To learn more about datetime-like frequency strings, please see [this link](#).

### Examples

Numeric `start` and `end` is supported.

```
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5])
              closed='right', dtype='interval[int64]')
```

Additionally, datetime-like input is also supported.

```
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
                      end=pd.Timestamp('2017-01-04'))
IntervalIndex([(2017-01-01, 2017-01-02], (2017-01-02, 2017-01-03],
              (2017-01-03, 2017-01-04]]
              closed='right', dtype='interval[datetime64[ns]]')
```

The `freq` parameter specifies the frequency between the left and right endpoints of the individual intervals within the `IntervalIndex`. For numeric start and end, the frequency must also be numeric.

```
>>> pd.interval_range(start=0, periods=4, freq=1.5)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]]
              closed='right', dtype='interval[float64]')
```

Similarly, for datetime-like start and end, the frequency must be convertible to a `DateOffset`.

```
>>> pd.interval_range(start=pd.Timestamp('2017-01-01'),
                      periods=3, freq='MS')
IntervalIndex([(2017-01-01, 2017-02-01], (2017-02-01, 2017-03-01],
              (2017-03-01, 2017-04-01]]
              closed='right', dtype='interval[datetime64[ns]]')
```

Specify start, end, and periods; the frequency is generated automatically (linearly spaced).

```
>>> pd.interval_range(start=0, end=6, periods=4)
IntervalIndex([(0.0, 1.5], (1.5, 3.0], (3.0, 4.5], (4.5, 6.0]]
              closed='right',
              dtype='interval[float64]')
```

The `closed` parameter specifies which endpoints of the individual intervals within the `IntervalIndex` are closed.

```
>>> pd.interval_range(end=5, periods=4, closed='both')
IntervalIndex([[1, 2], [2, 3], [3, 4], [4, 5]]
              closed='both', dtype='interval[int64]')
```

## 34.2.6 Top-level evaluation

---

<code>eval(expr[, parser, engine, truediv, ...])</code>	Evaluate a Python expression as a string using various backends.
---	--

---

### 34.2.6.1 pandas.eval

`pandas.eval(expr, parser='pandas', engine=None, truediv=True, local_dict=None, global_dict=None, resolvers=(), level=0, target=None, inplace=False)`  
 Evaluate a Python expression as a string using various backends.

The following arithmetic operations are supported: `+`, `-`, `*`, `/`, `**`, `%`, `//` (python engine only) along with the following boolean operations: `|` (or), `&` (and), and `~` (not). Additionally, the 'pandas' parser allows the use of `and`, `or`, and `not` with the same semantics as the corresponding bitwise operators. *Series* and *DataFrame* objects are supported and behave as they would with plain ol' Python evaluation.

**Parameters** `expr` : str or unicode

The expression to evaluate. This string cannot contain any Python *statements*, only

Python [expressions](#).

**parser** : string, default 'pandas', {'pandas', 'python'}

The parser to use to construct the syntax tree from the expression. The default of 'pandas' parses code slightly different than standard Python. Alternatively, you can parse an expression using the 'python' parser to retain strict Python semantics. See the [enhancing performance](#) documentation for more details.

**engine** : string or None, default 'numexpr', {'python', 'numexpr'}

The engine used to evaluate the expression. Supported engines are

- **None** : tries to use `numexpr`, falls back to `python`
- **'numexpr'** : **This default engine evaluates pandas objects using `numexpr`** for large speed ups in complex expressions with large frames.
- **'python'** : **Performs operations as if you had `eval`'d in top** level python. This engine is generally not that useful.

More backends may be available in the future.

**truediv** : bool, optional

Whether to use true division, like in Python `>= 3`

**local\_dict** : dict or None, optional

A dictionary of local variables, taken from `locals()` by default.

**global\_dict** : dict or None, optional

A dictionary of global variables, taken from `globals()` by default.

**resolvers** : list of dict-like or None, optional

A list of objects implementing the `__getitem__` special method that you can use to inject an additional collection of namespaces to use for variable lookup. For example, this is used in the [query\(\)](#) method to inject the `DataFrame.index` and `DataFrame.columns` variables that refer to their respective [DataFrame](#) instance attributes.

**level** : int, optional

The number of prior stack frames to traverse and add to the current scope. Most users will **not** need to change this parameter.

**target** : object, optional, default None

This is the target object for assignment. It is used when there is variable assignment in the expression. If so, then *target* must support item assignment with string keys, and if a copy is being returned, it must also support `.copy()`.

**inplace** : bool, default False

If *target* is provided, and the expression mutates *target*, whether to modify *target* in-place. Otherwise, return a copy of *target* with the mutation.

## Returns

**ndarray, numeric scalar, DataFrame, Series**

## Raises ValueError

There are many instances where such an error can be raised:



- *target=None*, but the expression is multiline.
- The expression is multiline, but not all them have item assignment. An example of such an arrangement is this:  

$$a = b + 1 \quad a + 2$$

Here, there are expressions on different lines, making it multiline, but the last line has no variable assigned to the output of  $a + 2$ .
- *inplace=True*, but the expression is missing item assignment.
- Item assignment is provided, but the *target* does not support string item assignment.
- Item assignment is provided and *inplace=False*, but the *target* does not support the *.copy()* method

See also:

`pandas.DataFrame.query`, `pandas.DataFrame.eval`

## Notes

The `dtype` of any objects involved in an arithmetic `%` operation are recursively cast to `float64`.

See the [enhancing performance](#) documentation for more details.

## 34.2.7 Testing

---

```
test([extra_args])
```

---

### 34.2.7.1 pandas.test

`pandas.test` (*extra\_args=None*)

## 34.3 Series

### 34.3.1 Constructor

---

<code>Series([data, index, dtype, name, copy, ...])</code>	One-dimensional ndarray with axis labels (including time series).
--	---

---

#### 34.3.1.1 pandas.Series

**class** `pandas.Series` (*data=None, index=None, dtype=None, name=None, copy=False, fastpath=False*)

One-dimensional ndarray with axis labels (including time series).

Labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index. Statistical methods from ndarray have been overridden to automatically exclude missing data (currently represented as NaN).

Operations between Series (+, -, /, \*) align values based on their associated index values— they need not be the

same length. The result index will be the sorted union of the two indexes.

**Parameters** **data** : array-like, dict, or scalar value

Contains data stored in Series

Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

**index** : array-like or Index (1d)

Values must be hashable and have the same length as *data*. Non-unique index values are allowed. Will default to RangeIndex (0, 1, 2, ..., n) if not provided. If both a dict and index sequence are used, the index will override the keys found in the dict.

**dtype** : numpy.dtype or None

If None, dtype will be inferred

**copy** : boolean, default False

Copy input data

### Attributes

<i>T</i>	return the transpose, which is by definition self
<i>asobject</i>	Return object Series which contains boxed values.
<i>at</i>	Access a single value for a row/column label pair.
<i>axes</i>	Return a list of the row axis labels
<i>base</i>	return the base object if the memory of the underlying data is shared
<i>blocks</i>	(DEPRECATED) Internal property, property synonym for <i>as_blocks()</i>
<i>data</i>	return the data pointer of the underlying data
<i>dtype</i>	return the dtype object of the underlying data
<i>dtypes</i>	return the dtype object of the underlying data
<i>flags</i>	
<i>ftype</i>	return if the data is sparsedense
<i>ftypes</i>	return if the data is sparsedense
<i>hasnans</i>	return if I have any nans; enables various perf speedups
<i>iat</i>	Access a single value for a row/column pair by integer position.
<i>iloc</i>	Purely integer-location based indexing for selection by position.
<i>index</i>	The index (axis labels) of the Series.
<i>is_monotonic</i>	Return boolean if values in the object are monotonic_increasing
<i>is_monotonic_decreasing</i>	Return boolean if values in the object are monotonic_decreasing
<i>is_monotonic_increasing</i>	Return boolean if values in the object are monotonic_increasing
<i>is_unique</i>	Return boolean if values in the object are unique
<i>itemsize</i>	return the size of the dtype of the item of the underlying data

Continued on next page

Table 24 – continued from previous page

<code>ix</code>	A primarily label-location based indexer, with integer position fallback.
<code>loc</code>	Access a group of rows and columns by label(s) or a boolean array.
<code>nbytes</code>	return the number of bytes in the underlying data
<code>ndim</code>	return the number of dimensions of the underlying data, by definition 1
<code>shape</code>	return a tuple of the shape of the underlying data
<code>size</code>	return the number of elements in the underlying data
<code>strides</code>	return the strides of the underlying data
<code>values</code>	Return Series as ndarray or ndarray-like depending on the dtype

**pandas.Series.T**`Series.T`

return the transpose, which is by definition self

**pandas.Series.asobject**`Series.asobject`

Return object Series which contains boxed values.

Deprecated since version 0.23.0: Use `astype(object)` instead.*this is an internal non-public method***pandas.Series.at**`Series.at`

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.**Raises KeyError**

When label does not exist in DataFrame

**See also:****`DataFrame.iat`** Access a single value for a row/column pair by integer position**`DataFrame.loc`** Access a group of rows and columns by label(s)**`Series.at`** Access a single value using a label**Examples**

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
```

(continues on next page)

(continued from previous page)

	A	B	C
4	0	2	3
5	0	4	1
6	10	20	30

Get value at specified row/column pair

```
>>> df.at[4, 'B']  
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10  
>>> df.at[4, 'B']  
10
```

Get value within a Series

```
>>> df.loc[5].at['B']  
4
```

## pandas.Series.axes

**Series.axes**

Return a list of the row axis labels

## pandas.Series.base

**Series.base**

return the base object if the memory of the underlying data is shared

## pandas.Series.blocks

**Series.blocks**

Internal property, property synonym for `as_blocks()`

Deprecated since version 0.21.0.

## pandas.Series.data

**Series.data**

return the data pointer of the underlying data

## pandas.Series.dtype

**Series.dtype**

return the dtype object of the underlying data

### **pandas.Series.dtypes**

#### **Series.dtypes**

return the dtype object of the underlying data

### **pandas.Series.flags**

#### **Series.flags**

### **pandas.Series.ftype**

#### **Series.ftype**

return if the data is sparsedense

### **pandas.Series.ftypes**

#### **Series.ftypes**

return if the data is sparsedense

### **pandas.Series.hasnans**

#### **Series.hasnans**

return if I have any nans; enables various perf speedups

### **pandas.Series.iat**

#### **Series.iat**

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

#### **Raises IndexError**

When integer position is out of bounds

#### **See also:**

**DataFrame.at** Access a single value for a row/column label pair

**DataFrame.loc** Access a group of rows and columns by label(s)

**DataFrame.iloc** Access a group of rows and columns by integer position(s)

### **Examples**

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                     columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

## pandas.Series.iloc

### Series.iloc

Purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

.iloc will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at [Selection by Position](#)

## pandas.Series.index

### Series.index

The index (axis labels) of the Series.

### **pandas.Series.is\_monotonic**

#### **Series.is\_monotonic**

Return boolean if values in the object are monotonic\_increasing

New in version 0.19.0.

#### **Returns**

**is\_monotonic** [boolean]

### **pandas.Series.is\_monotonic\_decreasing**

#### **Series.is\_monotonic\_decreasing**

Return boolean if values in the object are monotonic\_decreasing

New in version 0.19.0.

#### **Returns**

**is\_monotonic\_decreasing** [boolean]

### **pandas.Series.is\_monotonic\_increasing**

#### **Series.is\_monotonic\_increasing**

Return boolean if values in the object are monotonic\_increasing

New in version 0.19.0.

#### **Returns**

**is\_monotonic** [boolean]

### **pandas.Series.is\_unique**

#### **Series.is\_unique**

Return boolean if values in the object are unique

#### **Returns**

**is\_unique** [boolean]

### **pandas.Series.itemsize**

#### **Series.itemsize**

return the size of the dtype of the item of the underlying data

### **pandas.Series.ix**

#### **Series.ix**

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, **ONLY** label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at [Advanced Indexing](#).

## pandas.Series.loc

### Series.loc

Access a group of rows and columns by label(s) or a boolean array.

`.loc[]` is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a' : 'f'.

**Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at [Selection by Label](#)

#### Raises **KeyError**:

when any items are not found

See also:

**DataFrame.at** Access a single value for a row/column label pair

**DataFrame.iloc** Access group of rows and columns by integer position(s)

**DataFrame.xs** Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

**Series.loc** Access group of values using labels

## Examples

### Getting values



```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
```

	max_speed	shield
viper	4	5
sidewinder	7	8

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
```

	max_speed	shield
sidewinder	7	8

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
```

	max_speed	shield
sidewinder	7	8

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
```

	max_speed
sidewinder	7

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
           max_speed  shield
sidewinder           7       8
```

### Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
           max_speed  shield
cobra                1       2
viper                4      50
sidewinder           7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
           max_speed  shield
cobra             10      10
viper              4      50
sidewinder         7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
           max_speed  shield
cobra             30      10
viper             30      50
sidewinder        30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
           max_speed  shield
cobra             30      10
viper              0       0
sidewinder         0       0
```

### Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
           max_speed  shield
7                1       2
8                4       5
9                7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
      max_speed  shield
7             1       2
8             4       5
9             7       8
```

### Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
```

		max_speed	shield
cobra	mark i	12	2
	mark ii	0	4
sidewinder	mark i	10	20
	mark ii	1	4
viper	mark ii	7	1
	mark iii	16	36

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
      max_speed  shield
mark i         12       2
mark ii        0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield       2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['cobra', 'mark ii']]
      max_speed  shield
cobra mark ii         0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
      max_speed  shield
cobra      mark i      12      2
           mark ii       0      4
sidewinder mark i      10     20
           mark ii       1      4
viper      mark ii       7      1
           mark iii     16     36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):(('viper', 'mark ii'))]
      max_speed  shield
cobra      mark i      12      2
           mark ii       0      4
sidewinder mark i      10     20
           mark ii       1      4
viper      mark ii       7      1
```

## **pandas.Series.nbytes**

### **Series.nbytes**

return the number of bytes in the underlying data

## **pandas.Series.ndim**

### **Series.ndim**

return the number of dimensions of the underlying data, by definition 1

## **pandas.Series.shape**

### **Series.shape**

return a tuple of the shape of the underlying data

## **pandas.Series.size**

### **Series.size**

return the number of elements in the underlying data

## **pandas.Series.strides**

### **Series.strides**

return the strides of the underlying data

## pandas.Series.values

### Series.values

Return Series as ndarray or ndarray-like depending on the dtype

#### Returns

**arr** [numpy.ndarray or ndarray-like]

### Examples

```
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```
>>> pd.Series(list('aabc')).astype('category').values
[a, a, b, c]
Categories (3, object): [a, b, c]
```

Timezone aware datetime data is converted to UTC:

```
>>> pd.Series(pd.date_range('20130101', periods=3,
...                          tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
       '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

<b>empty</b>	
<b>imag</b>	
<b>is_copy</b>	
<b>name</b>	
<b>real</b>	

## Methods

<i>abs()</i>	Return a Series/DataFrame with absolute numeric value of each element.
<i>add</i> (other[, level, fill_value, axis])	Addition of series and other, element-wise (binary operator <i>add</i> ).
<i>add_prefix</i> (prefix)	Prefix labels with string <i>prefix</i> .
<i>add_suffix</i> (suffix)	Suffix labels with string <i>suffix</i> .
<i>agg</i> (func[, axis])	Aggregate using one or more operations over the specified axis.
<i>aggregate</i> (func[, axis])	Aggregate using one or more operations over the specified axis.
<i>align</i> (other[, join, axis, level, copy, ...])	Align two objects on their axes with the specified join method for each axis Index

Continued on next page

Table 25 – continued from previous page

<code>all([axis, bool_only, skipna, level])</code>	Return whether all elements are True, potentially over an axis.
<code>any([axis, bool_only, skipna, level])</code>	Return whether any element is True over requested axis.
<code>append(to_append[, ignore_index, ...])</code>	Concatenate two or more Series.
<code>apply(func[, convert_dtype, args])</code>	Invoke function on values of Series.
<code>argmax([axis, skipna])</code>	(DEPRECATED) ..
<code>argmin([axis, skipna])</code>	(DEPRECATED) ..
<code>argsort([axis, kind, order])</code>	Overrides ndarray.argsort.
<code>as_blocks([copy])</code>	(DEPRECATED) Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
<code>as_matrix([columns])</code>	(DEPRECATED) Convert the frame to its Numpy-array representation.
<code>asfreq(freq[, method, how, normalize, ...])</code>	Convert TimeSeries to specified frequency.
<code>asof(when[, subset])</code>	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
<code>astype(dtype[, copy, errors])</code>	Cast a pandas object to a specified dtype dtype.
<code>at_time(time[, asof])</code>	Select values at particular time of day (e.g.
<code>autocorr([lag])</code>	Lag-N autocorrelation
<code>between(left, right[, inclusive])</code>	Return boolean Series equivalent to left <= series <= right.
<code>between_time(start_time, end_time[, ...])</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill([axis, inplace, limit, downcast])</code>	Synonym for <code>DataFrame.fillna(method='bfill')</code>
<code>bool()</code>	Return the bool of a single element PandasObject.
<code>cat</code>	alias of <code>pandas.core.arrays.categorical.CategoricalAccessor</code>
<code>clip([lower, upper, axis, inplace])</code>	Trim values at input threshold(s).
<code>clip_lower(threshold[, axis, inplace])</code>	Return copy of the input with values below a threshold truncated.
<code>clip_upper(threshold[, axis, inplace])</code>	Return copy of input with values above given value(s) truncated.
<code>combine(other, func[, fill_value])</code>	Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other
<code>combine_first(other)</code>	Combine Series values, choosing the calling Series's values first.
<code>compound([axis, skipna, level])</code>	Return the compound percentage of the values for the requested axis
<code>compress(condition, *args, **kwargs)</code>	Return selected slices of an array along given axis as a Series
<code>consolidate([inplace])</code>	(DEPRECATED) Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray).
<code>convert_objects([convert_dates, ...])</code>	(DEPRECATED) Attempt to infer better dtype for object columns.
<code>copy([deep])</code>	Make a copy of this object's indices and data.

Continued on next page

Table 25 – continued from previous page

<code>corr(other[, method, min_periods])</code>	Compute correlation with <i>other</i> Series, excluding missing values
<code>count([level])</code>	Return number of non-NA/null observations in the Series
<code>cov(other[, min_periods])</code>	Compute covariance with Series, excluding missing values
<code>cummax([axis, skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin([axis, skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod([axis, skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>cumsum([axis, skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>describe([percentiles, include, exclude])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>diff([periods])</code>	First discrete difference of element.
<code>div(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>divide(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>divmod(other[, level, fill_value, axis])</code>	Integer division and modulo of series and other, element-wise (binary operator <i>divmod</i> ).
<code>dot(other)</code>	Matrix multiplication with DataFrame or inner-product with Series objects.
<code>drop([labels, axis, index, columns, level, ...])</code>	Return Series with specified index labels removed.
<code>drop_duplicates([keep, inplace])</code>	Return Series with duplicate values removed.
<code>dropna([axis, inplace])</code>	Return a new Series with missing values removed.
<code>dt</code>	alias of <code>pandas.core.indexes.accessors.CombinedDatetimelikeProperties</code>
<code>uplicated([keep])</code>	Indicate duplicate Series values.
<code>eq(other[, level, fill_value, axis])</code>	Equal to of series and other, element-wise (binary operator <i>eq</i> ).
<code>equals(other)</code>	Determines if two NDFrame objects contain the same elements.
<code>ewm([com, span, halflife, alpha, ...])</code>	Provides exponential weighted functions
<code>expanding([min_periods, center, axis])</code>	Provides expanding transformations.
<code>factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>ffill([axis, inplace, limit, downcast])</code>	Synonym for <code>DataFrame.fillna(method='ffill')</code>
<code>fillna([value, method, axis, inplace, ...])</code>	Fill NA/NaN values using the specified method
<code>filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>first_valid_index()</code>	Return index for first non-NA/null value.
<code>floordiv(other[, level, fill_value, axis])</code>	Integer division of series and other, element-wise (binary operator <i>floordiv</i> ).

Continued on next page

Table 25 – continued from previous page

<code>from_array(arr[, index, name, dtype, copy, ...])</code>	Construct Series from array.
<code>from_csv(path[, sep, parse_dates, header, ...])</code>	(DEPRECATED) Read CSV file.
<code>ge(other[, level, fill_value, axis])</code>	Greater than or equal to of series and other, element-wise (binary operator <i>ge</i> ).
<code>get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>get_dtype_counts()</code>	Return counts of unique dtypes in this object.
<code>get_ftype_counts()</code>	(DEPRECATED) Return counts of unique ftypes in this object.
<code>get_value(label[, takeable])</code>	(DEPRECATED) Quickly retrieve single value at passed index label
<code>get_values()</code>	same as values (but handles sparseness conversions); is a view
<code>groupby([by, axis, level, as_index, sort, ...])</code>	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
<code>gt(other[, level, fill_value, axis])</code>	Greater than of series and other, element-wise (binary operator <i>gt</i> ).
<code>head([n])</code>	Return the first <i>n</i> rows.
<code>hist([by, ax, grid, xlabelsize, xrot, ...])</code>	Draw histogram of the input series using matplotlib
<code>idxmax([axis, skipna])</code>	Return the row label of the maximum value.
<code>idxmin([axis, skipna])</code>	Return the row label of the minimum value.
<code>infer_objects()</code>	Attempt to infer better dtypes for object columns.
<code>interpolate([method, axis, limit, inplace, ...])</code>	Interpolate values according to different methods.
<code>isin(values)</code>	Check whether <i>values</i> are contained in Series.
<code>isna()</code>	Detect missing values.
<code>isnull()</code>	Detect missing values.
<code>item()</code>	return the first element of the underlying data as a python scalar
<code>items()</code>	Lazily iterate over (index, value) tuples
<code>iteritems()</code>	Lazily iterate over (index, value) tuples
<code>keys()</code>	Alias for index
<code>kurt([axis, skipna, level, numeric_only])</code>	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>kurtosis([axis, skipna, level, numeric_only])</code>	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>last(offset)</code>	Convenience method for subsetting final periods of time series data based on a date offset.
<code>last_valid_index()</code>	Return index for last non-NA/null value.
<code>le(other[, level, fill_value, axis])</code>	Less than or equal to of series and other, element-wise (binary operator <i>le</i> ).
<code>lt(other[, level, fill_value, axis])</code>	Less than of series and other, element-wise (binary operator <i>lt</i> ).
<code>mad([axis, skipna, level])</code>	Return the mean absolute deviation of the values for the requested axis
<code>map(arg[, na_action])</code>	Map values of Series using input correspondence (a dict, Series, or function).

Continued on next page



Table 25 – continued from previous page

<code>mask(cond[, other, inplace, axis, level, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>max([axis, skipna, level, numeric_only])</code>	This method returns the maximum of the values in the object.
<code>mean([axis, skipna, level, numeric_only])</code>	Return the mean of the values for the requested axis
<code>median([axis, skipna, level, numeric_only])</code>	Return the median of the values for the requested axis
<code>memory_usage([index, deep])</code>	Return the memory usage of the Series.
<code>min([axis, skipna, level, numeric_only])</code>	This method returns the minimum of the values in the object.
<code>mod(other[, level, fill_value, axis])</code>	Modulo of series and other, element-wise (binary operator <i>mod</i> ).
<code>mode()</code>	Return the mode(s) of the dataset.
<code>mul(other[, level, fill_value, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>multiply(other[, level, fill_value, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>ne(other[, level, fill_value, axis])</code>	Not equal to of series and other, element-wise (binary operator <i>ne</i> ).
<code>nlargest([n, keep])</code>	Return the largest <i>n</i> elements.
<code>nonzero()</code>	Return the <i>integer</i> indices of the elements that are non-zero
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nsmallest([n, keep])</code>	Return the smallest <i>n</i> elements.
<code>nunique([dropna])</code>	Return number of unique elements in the object.
<code>pct_change([periods, fill_method, limit, freq])</code>	Percentage change between the current and a prior element.
<code>pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code>
<code>plot</code>	alias of <code>pandas.plotting._core.SeriesPlotMethods</code>
<code>pop(item)</code>	Return item and drop from frame.
<code>pow(other[, level, fill_value, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>pow</i> ).
<code>prod([axis, skipna, level, numeric_only, ...])</code>	Return the product of the values for the requested axis
<code>product([axis, skipna, level, numeric_only, ...])</code>	Return the product of the values for the requested axis
<code>ptp([axis, skipna, level, numeric_only])</code>	Returns the difference between the maximum value and the minimum value in the object.
<code>put(*args, **kwargs)</code>	Applies the <i>put</i> method to its <i>values</i> attribute if it has one.
<code>quantile([q, interpolation])</code>	Return value at the given quantile, a la <code>numpy.percentile</code> .
<code>radd(other[, level, fill_value, axis])</code>	Addition of series and other, element-wise (binary operator <i>radd</i> ).
<code>rank([axis, method, numeric_only, ...])</code>	Compute numerical data ranks (1 through <i>n</i> ) along axis.
<code>ravel([order])</code>	Return the flattened underlying data as an ndarray

Continued on next page

Table 25 – continued from previous page

<code>rdiv(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<code>reindex([index])</code>	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_axis(labels[, axis])</code>	(DEPRECATED) Conform Series to new index with optional filling logic.
<code>reindex_like(other[, method, copy, limit, ...])</code>	Return an object with matching indices to myself.
<code>rename([index])</code>	Alter Series index labels or name
<code>rename_axis(mapper[, axis, copy, inplace])</code>	Alter the name of the index or columns.
<code>reorder_levels(order)</code>	Rearrange index levels using input order.
<code>repeat(repeats, *args, **kwargs)</code>	Repeat elements of an Series.
<code>replace([to_replace, value, inplace, limit, ...])</code>	Replace values given in <i>to_replace</i> with <i>value</i> .
<code>resample(rule[, how, axis, fill_method, ...])</code>	Convenience method for frequency conversion and resampling of time series.
<code>reset_index([level, drop, name, inplace])</code>	Generate a new DataFrame or Series with the index reset.
<code>rfloordiv(other[, level, fill_value, axis])</code>	Integer division of series and other, element-wise (binary operator <i>rfloordiv</i> ).
<code>rmod(other[, level, fill_value, axis])</code>	Modulo of series and other, element-wise (binary operator <i>rmod</i> ).
<code>rmul(other[, level, fill_value, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>rmul</i> ).
<code>rolling(window[, min_periods, center, ...])</code>	Provides rolling window calculations.
<code>round([decimals])</code>	Round each value in a Series to the given number of decimals.
<code>rpow(other[, level, fill_value, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>rpow</i> ).
<code>rsub(other[, level, fill_value, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>rsub</i> ).
<code>rtruediv(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<code>sample([n, frac, replace, weights, ...])</code>	Return a random sample of items from an axis of object.
<code>searchsorted(value[, side, sorter])</code>	Find indices where elements should be inserted to maintain order.
<code>select(crit[, axis])</code>	(DEPRECATED) Return data corresponding to axis labels matching criteria
<code>sem([axis, skipna, level, ddof, numeric_only])</code>	Return unbiased standard error of the mean over requested axis.
<code>set_axis(labels[, axis, inplace])</code>	Assign desired index to given axis.
<code>set_value(label, value[, takeable])</code>	(DEPRECATED) Quickly set single value at passed label.
<code>shift([periods, freq, axis])</code>	Shift index by desired number of periods with an optional time freq
<code>skew([axis, skipna, level, numeric_only])</code>	Return unbiased skew over requested axis Normalized by N-1
<code>slice_shift([periods, axis])</code>	Equivalent to <i>shift</i> without copying data.
<code>sort_index([axis, level, ascending, ...])</code>	Sort Series by index labels.
<code>sort_values([axis, ascending, inplace, ...])</code>	Sort by the values.

Continued on next page

Table 25 – continued from previous page

<code>sortlevel([level, ascending, sort_remaining])</code>	(DEPRECATED) Sort Series with MultiIndex by chosen level.
<code>squeeze([axis])</code>	Squeeze length 1 dimensions.
<code>std([axis, skipna, level, ddof, numeric_only])</code>	Return sample standard deviation over requested axis.
<code>str</code>	alias of <code>pandas.core.strings.StringMethods</code>
<code>sub(other[, level, fill_value, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>subtract(other[, level, fill_value, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>sum([axis, skipna, level, numeric_only, ...])</code>	Return the sum of the values for the requested axis
<code>swapaxes(axis1, axis2[, copy])</code>	Interchange axes and swap values axes appropriately
<code>swaplevel([i, j, copy])</code>	Swap levels i and j in a MultiIndex
<code>tail([n])</code>	Return the last <i>n</i> rows.
<code>take(indices[, axis, convert, is_copy])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>to_csv([path, index, sep, na_rep, ...])</code>	Write Series to a comma-separated values (csv) file
<code>to_dense()</code>	Return dense representation of NDFrame (as opposed to sparse)
<code>to_dict([into])</code>	Convert Series to {label -> value} dict or dict-like object.
<code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code>	Write Series to an excel sheet
<code>to_frame([name])</code>	Convert Series to DataFrame
<code>to_hdf(path_or_buf, key, **kwargs)</code>	Write the contained data to an HDF5 file using HDF-Store.
<code>to_json([path_or_buf, orient, date_format, ...])</code>	Convert the object to a JSON string.
<code>to_latex([buf, columns, col_space, header, ...])</code>	Render an object to a tabular environment table.
<code>to_msgpack([path_or_buf, encoding])</code>	msgpack (serialize) object to input file path
<code>to_period([freq, copy])</code>	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)
<code>to_pickle(path[, compression, protocol])</code>	Pickle (serialize) object to file.
<code>to_sparse([kind, fill_value])</code>	Convert Series to SparseSeries
<code>to_sql(name, con[, schema, if_exists, ...])</code>	Write records stored in a DataFrame to a SQL database.
<code>to_string([buf, na_rep, float_format, ...])</code>	Render a string representation of the Series
<code>to_timestamp([freq, how, copy])</code>	Cast to datetimeindex of timestamps, at <i>beginning</i> of period
<code>to_xarray()</code>	Return an xarray object from the pandas object.
<code>tolist()</code>	Return a list of the values.
<code>transform(func, *args, **kwargs)</code>	Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
<code>transpose(*args, **kwargs)</code>	return the transpose, which is by definition self
<code>truediv(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.

Continued on next page

Table 25 – continued from previous page

<code>tshift([periods, freq, axis])</code>	Shift the time index, using the index's frequency if available.
<code>tz_convert(tz[, axis, level, copy])</code>	Convert tz-aware axis to target time zone.
<code>tz_localize(tz[, axis, level, copy, ambiguous])</code>	Localize tz-naive TimeSeries to target time zone.
<code>unique()</code>	Return unique values of Series object.
<code>unstack([level, fill_value])</code>	Unstack, a.k.a.
<code>update(other)</code>	Modify Series in place using non-NA values from passed Series.
<code>valid([inplace])</code>	(DEPRECATED) Return Series without null values.
<code>value_counts([normalize, sort, ascending, ...])</code>	Returns object containing counts of unique values.
<code>var([axis, skipna, level, ddof, numeric_only])</code>	Return unbiased variance over requested axis.
<code>view([dtype])</code>	Create a new view of the Series.
<code>where(cond[, other, inplace, axis, level, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .
<code>xs(key[, axis, level, drop_level])</code>	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

## pandas.Series.abs

`Series.abs()`

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

### Returns abs

Series/DataFrame containing the absolute value of each element.

### See also:

`numpy.absolute` calculate the absolute value element-wise.

### Notes

For complex inputs,  $1.2 + 1j$ , the absolute value is  $\sqrt{a^2 + b^2}$ .

### Examples

Absolute numeric values in a Series.

```

>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64

```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0    1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using `argsort` (from [StackOverflow](#)).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
   a  b  c
0  4 10 100
1  5 20  50
2  6 30 -30
3  7 40 -50
>>> df.loc[(df.c - 43).abs().argsort()]
   a  b  c
1  5 20  50
0  4 10 100
2  6 30 -30
3  7 40 -50
```

## pandas.Series.add

`Series.add(other, level=None, fill_value=None, axis=0)`

Addition of series and other, element-wise (binary operator `add`).

Equivalent to `series + other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

**See also:***Series.radd***Examples**

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d     NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b     NaN
d    1.0
e     NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e     NaN
dtype: float64
```

**pandas.Series.add\_prefix****Series.add\_prefix** (*prefix*)Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters prefix** : str

The string to add before each label.

**Returns Series or DataFrame**

New Series or DataFrame with updated labels.

**See also:***Series.add\_suffix* Suffix row labels with string *suffix*.*DataFrame.add\_suffix* Suffix column labels with string *suffix*.**Examples**

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
   col_A  col_B
0      1      3
1      2      4
2      3      5
3      4      6
```

### pandas.Series.add\_suffix

`Series.add_suffix(suffix)`  
 Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

**Parameters** `suffix` : str

The string to add after each label.

**Returns** **Series or DataFrame**

New Series or DataFrame with updated labels.

**See also:**

**`Series.add_prefix`** Prefix row labels with string *prefix*.

**`DataFrame.add_prefix`** Prefix column labels with string *prefix*.

### Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
```

(continues on next page)

(continued from previous page)

```
3      4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item      1
1_item      2
2_item      3
3_item      4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
   A_col  B_col
0      1      3
1      2      4
2      3      5
3      4      6
```

## pandas.Series.agg

`Series.agg(func, axis=0, *args, **kwargs)`

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

**Parameters** **func** : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**axis** : {0 or 'index'}

Parameter needed for compatibility with DataFrame.

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.



**Returns****aggregated** [Series]**See also:***pandas.Series.apply*, *pandas.Series.transform***Notes***agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

**Examples**

```
>>> s = Series(np.random.randn(10))
```

```
>>> s.agg('min')
-1.3018049988556679
```

```
>>> s.agg(['min', 'max'])
min    -1.301805
max     1.127688
dtype: float64
```

**pandas.Series.aggregate****Series.aggregate** (*func*, *axis=0*, *\*args*, *\*\*kwargs*)

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

**Parameters** **func** : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**axis** : {0 or 'index'}

Parameter needed for compatibility with DataFrame.

**\*args**Positional arguments to pass to *func*.**\*\*kwargs**

Keyword arguments to pass to *func*.

### Returns

**aggregated** [Series]

### See also:

[`pandas.Series.apply`](#), [`pandas.Series.transform`](#)

### Notes

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

### Examples

```
>>> s = Series(np.random.randn(10))
```

```
>>> s.agg('min')
-1.3018049988556679
```

```
>>> s.agg(['min', 'max'])
min    -1.301805
max     1.127688
dtype: float64
```

## pandas.Series.align

`Series.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0, broadcast_axis=None)`

Align two objects on their axes with the specified join method for each axis Index

### Parameters

**other** [DataFrame or Series]

**join** [{ 'outer', 'inner', 'left', 'right' }, default 'outer']

**axis** : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

**level** : int or level name, default None

Broadcast across a level, matching Index values on the passed MultiIndex level

**copy** : boolean, default True

Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any “compatible” value

**method** [str, default None]

**limit** [int, default None]

**fill\_axis** : {0 or 'index'}, default 0

Filling axis, method and limit

**broadcast\_axis** : {0 or 'index'}, default None

Broadcast values along this axis, if aligning two objects of different dimensions

**Returns** (**left, right**) : (Series, type of other)

Aligned objects

## pandas.Series.all

`Series.all` (*axis=0, bool\_only=None, skipna=True, level=None, \*\*kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

**Parameters** **axis** : {0 or 'index', 1 or 'columns', None}, default 0

Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**\*\*kwargs** : any, default None

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**all** [scalar or Series (if level specified)]

**See also:**

[`pandas.Series.all`](#) Return True if all elements are True

[`pandas.DataFrame.any`](#) Return True if one (or more) elements are True

## Examples

### Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

### DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1  col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1    True
col2   False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

## pandas.Series.any

`Series.any` (*axis=0, bool\_only=None, skipna=True, level=None, \*\*kwargs*)

Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

**Parameters** `axis`: {0 or 'index', 1 or 'columns', None}, default 0

Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar.

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**\*\*kwargs** : any, default None

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**any** [scalar or Series (if level specified)]

### See also:

[`pandas.DataFrame.all`](#) Return whether all elements are True.

## Examples

### Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

### DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A      True
B      True
C     False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
   A  B
0  True  1
1 False  2
```

```
>>> df.any(axis='columns')
0    True
1    True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
   A  B
0  True  1
1 False  0
```

```
>>> df.any(axis='columns')
0    True
1   False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

`any` for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

## pandas.Series.append

`Series.append(to_append, ignore_index=False, verify_integrity=False)`  
Concatenate two or more Series.

### Parameters

**to\_append** [Series or list/tuple of Series]

**ignore\_index** : boolean, default False

If True, do not use the index labels.

New in version 0.19.0.

**verify\_integrity** : boolean, default False

If True, raise Exception on creating index with duplicates

### Returns

**appended** [Series]

**See also:**

[`pandas.concat`](#) General function to concatenate DataFrame, Series or Panel objects

### Notes

Iteratively appending to a Series can be more computationally intensive than a single concatenate. A better solution is to append values to a list and then concatenate the list with the original Series all at once.

## Examples

```
>>> s1 = pd.Series([1, 2, 3])
>>> s2 = pd.Series([4, 5, 6])
>>> s3 = pd.Series([4, 5, 6], index=[3,4,5])
>>> s1.append(s2)
0    1
1    2
2    3
0    4
1    5
2    6
dtype: int64
```

```
>>> s1.append(s3)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With `ignore_index` set to `True`:

```
>>> s1.append(s2, ignore_index=True)
0    1
1    2
2    3
3    4
4    5
5    6
dtype: int64
```

With `verify_integrity` set to `True`:

```
>>> s1.append(s2, verify_integrity=True)
Traceback (most recent call last):
...
ValueError: Indexes have overlapping values: [0, 1, 2]
```

## pandas.Series.apply

`Series.apply(func, convert_dtype=True, args=(), **kwargs)`

Invoke function on values of Series. Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values

### Parameters

**func** [function]

**convert\_dtype** : boolean, default True

Try to find better dtype for elementwise function results. If False, leave as dtype=object

**args** : tuple

Positional arguments to pass to function in addition to the value

Additional keyword arguments will be passed as keywords to the function

### Returns

y [Series or DataFrame if func returns a Series]

### See also:

*Series.map* For element-wise operations

*Series.agg* only perform aggregating type operations

*Series.transform* only perform transforming type operations

### Examples

Create a series with typical summer temperatures for each city.

```
>>> import pandas as pd
>>> import numpy as np
>>> series = pd.Series([20, 21, 12], index=['London',
... 'New York', 'Helsinki'])
>>> series
London      20
New York    21
Helsinki    12
dtype: int64
```

Square the values by defining a function and passing it as an argument to `apply()`.

```
>>> def square(x):
...     return x**2
>>> series.apply(square)
London      400
New York    441
Helsinki    144
dtype: int64
```

Square the values by passing an anonymous function as an argument to `apply()`.

```
>>> series.apply(lambda x: x**2)
London      400
New York    441
Helsinki    144
dtype: int64
```

Define a custom function that needs additional positional arguments and pass these additional arguments using the `args` keyword.

```
>>> def subtract_custom_value(x, custom_value):
...     return x-custom_value
```

```
>>> series.apply(subtract_custom_value, args=(5,))
London      15
New York    16
Helsinki     7
dtype: int64
```



Define a custom function that takes keyword arguments and pass these arguments to `apply`.

```
>>> def add_custom_values(x, **kwargs):
...     for month in kwargs:
...         x+=kwargs[month]
...     return x
```

```
>>> series.apply(add_custom_values, june=30, july=20, august=25)
London      95
New York    96
Helsinki    87
dtype: int64
```

Use a function from the Numpy library.

```
>>> series.apply(np.log)
London      2.995732
New York    3.044522
Helsinki    2.484907
dtype: float64
```

## pandas.Series.argmax

`Series.argmax` (*axis=0, skipna=True, \*args, \*\*kwargs*)

Deprecated since version 0.21.0:

**‘argmax’ is deprecated, use ‘idxmax’ instead.** The behavior of ‘argmax’ will be corrected to return the positional maximum in the future. Use ‘`series.values.argmax`’ to get the position of the maximum now.

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

**Parameters** `skipna` : boolean, default True

Exclude NA/null values. If the entire Series is NA, the result will be NA.

**axis** : int, default 0

For compatibility with `DataFrame.idxmax`. Redundant for application on Series.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

**idxmax** [Index of maximum of values.]

**Raises** `ValueError`

If the Series is empty.

**See also:**

**numpy.argmax** Return indices of the maximum values along the given axis.

**DataFrame.idxmax** Return index of first occurrence of maximum over requested axis.

**Series.idxmin** Return index *label* of the first occurrence of minimum of values.

## Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

## Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...               index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B     NaN
C     4.0
D     3.0
E     4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If `skipna` is `False` and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

## pandas.Series.argmin

`Series.argmax` (*axis=None, skipna=True, \*args, \*\*kwargs*)

Deprecated since version 0.21.0:

**‘argmin’ is deprecated, use ‘idxmin’ instead.** The behavior of **‘argmin’** will be corrected to return the positional minimum in the future. Use `‘series.values.argmin’` to get the position of the minimum now.

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

**Parameters** `skipna` : boolean, default True

Exclude NA/null values. If the entire Series is NA, the result will be NA.

**axis** : int, default 0

For compatibility with `DataFrame.idxmin`. Redundant for application on Series.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**idxmin** [Index of minimum of values.]

### Raises

**ValueError**

If the Series is empty.

**See also:**

**numpy.argmax** Return indices of the minimum values along the given axis.

**DataFrame.idxmin** Return index of first occurrence of minimum over requested axis.

**Series.idxmax** Return index *label* of the first occurrence of maximum of values.

## Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the minimum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

## Examples

```
>>> s = pd.Series(data=[1, None, 4, 1],
...                 index=['A', 'B', 'C', 'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If *skipna* is False and there is an NA value in the data, the function returns nan.

```
>>> s.idxmin(skipna=False)
nan
```

## pandas.Series.argsort

**Series.argsort** (*axis=0, kind='quicksort', order=None*)

Overrides `ndarray.argsort`. Argsorts the value, omitting NA/null values, and places the result in the same locations as the non-NA values

### Parameters

**axis** [int (can only be zero)]

**kind** : {'mergesort', 'quicksort', 'heapsort'}, default 'quicksort'

Choice of sorting algorithm. See `np.sort` for more information. 'mergesort' is the only stable algorithm

**order** [ignored]

### Returns

**argsorted** [Series, with -1 indicated where nan values are present]

### See also:

`numpy.ndarray.argsort`

### pandas.Series.as\_blocks

`Series.as_blocks (copy=True)`

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

**NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in `as_matrix`)**

#### Parameters

**copy** [boolean, default True]

#### Returns

**values** [a dict of dtype -> Constructor Types]

### pandas.Series.as\_matrix

`Series.as_matrix (columns=None)`

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.

#### Parameters **columns: list, optional, default:None**

If None, return all columns, otherwise, returns specified columns.

#### Returns **values** : ndarray

If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

#### See also:

`pandas.DataFrame.values`

#### Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By `numpy.find_common_type` convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use `‘.values’`.

### pandas.Series.asfreq

`Series.asfreq (freq, method=None, how=None, normalize=False, fill_value=None)`

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

### Parameters

**freq** [DateOffset object, or string]

**method** : { 'backfill'/'bfill', 'pad'/'ffill' }, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- 'pad' / 'ffill': propagate last valid observation forward to next valid
- 'backfill' / 'bfill': use NEXT valid observation to fill

**how** : { 'start', 'end' }, default end

For PeriodIndex only, see PeriodIndex.asfreq

**normalize** : bool, default False

Whether to reset output index to midnight

**fill\_value**: scalar, optional

Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

### Returns

**converted** [type of caller]

**See also:**

[`reindex`](#)

### Notes

To learn more about the frequency strings, please see [this link](#).

### Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
```

	s
2000-01-01 00:00:00	0.0
2000-01-01 00:01:00	NaN
2000-01-01 00:02:00	2.0
2000-01-01 00:03:00	3.0

Upsample the series into 30 second bins.

```
>>> df.iasfreq(freq='30S')
      S
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   NaN
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30   NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a fill value.

```
>>> df.iasfreq(freq='30S', fill_value=9.0)
      S
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00   NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a method.

```
>>> df.iasfreq(freq='30S', method='bfill')
      S
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

## pandas.Series.asof

`Series.asof` (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

### Parameters

**where** [date or array of dates]

**subset** : string or list of strings, default None

if not None use these columns for NaN propagation

### Returns where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

**where is Index:** same shape object as input

See also:

`merge_asof`

## Notes

Dates are assumed to be sorted Raises if this is not the case

## pandas.Series.astype

`Series.astype(dtype, copy=True, errors='raise', **kwargs)`

Cast a pandas object to a specified dtype dtype.

**Parameters** `dtype` : data type, or dict of column name -> data type

Use a `numpy.dtype` or Python type to cast entire pandas object to the same type. Alternatively, use `{col: dtype, ...}`, where `col` is a column label and `dtype` is a `numpy.dtype` or Python type to cast one or more of the DataFrame's columns to column-specific types.

**copy** : bool, default True.

Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

**errors** : { 'raise', 'ignore' }, default 'raise'.

Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

**raise\_on\_error** : raise on invalid input

Deprecated since version 0.20.0: Use `errors` instead

**kwargs** [keyword arguments to pass on to the constructor]

## Returns

**casted** [type of caller]

See also:

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to a numeric type.

`numpy.ndarray.astype` Cast a numpy array to a specified type.

## Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1 # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

## pandas.Series.at\_time

`Series.at_time(time, asof=False)`

Select values at particular time of day (e.g. 9:30AM).

### Parameters

**time** [datetime.time or string]

### Returns

**values\_at\_time** [type of caller]

### Raises `TypeError`

If the index is not a `DatetimeIndex`

**See also:**

**`between_time`** Select values between particular times of the day

**`first`** Select initial periods of time series based on a date offset



**last** Select final periods of time series based on a date offset

**DatetimeIndex.indexer\_at\_time** Get just the index locations for values at particular time of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09 00:00:00	1
2018-04-09 12:00:00	2
2018-04-10 00:00:00	3
2018-04-10 12:00:00	4

```
>>> ts.at_time('12:00')
```

	A
2018-04-09 12:00:00	2
2018-04-10 12:00:00	4

## pandas.Series.autocorr

**Series.autocorr** (*lag=1*)

Lag-N autocorrelation

**Parameters** **lag** : int, default 1

Number of lags to apply before performing autocorrelation.

**Returns**

**autocorr** [float]

## pandas.Series.between

**Series.between** (*left, right, inclusive=True*)

Return boolean Series equivalent to `left <= series <= right`.

This function returns a boolean vector containing *True* wherever the corresponding Series element is between the boundary values *left* and *right*. NA values are treated as *False*.

**Parameters** **left** : scalar

Left boundary.

**right** : scalar

Right boundary.

**inclusive** : bool, default True

Include boundaries.

**Returns** **Series**

Each element will be a boolean.

See also:

`pandas.Series.gt` Greater than of series and other

`pandas.Series.lt` Less than of series and other

## Notes

This function is equivalent to `(left <= ser) & (ser <= right)`

## Examples

```
>>> s = pd.Series([2, 0, 4, 8, np.nan])
```

Boundary values are included by default:

```
>>> s.between(1, 4)
0      True
1     False
2      True
3     False
4     False
dtype: bool
```

With *inclusive* set to `False` boundary values are excluded:

```
>>> s.between(1, 4, inclusive=False)
0      True
1     False
2     False
3     False
4     False
dtype: bool
```

*left* and *right* can be any scalar value:

```
>>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
>>> s.between('Anna', 'Daniel')
0     False
1      True
2      True
3     False
dtype: bool
```

## pandas.Series.between\_time

`Series.between_time` (*start\_time*, *end\_time*, *include\_start=True*, *include\_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting *start\_time* to be later than *end\_time*, you can get the times that are *not* between the two times.

### Parameters

**start\_time** [datetime.time or string]

**end\_time** [datetime.time or string]  
**include\_start** [boolean, default True]  
**include\_end** [boolean, default True]

**Returns**

**values\_between\_time** [type of caller]

**Raises TypeError**

If the index is not a *DatetimeIndex*

**See also:**

**at\_time** Select values at a particular time of the day

**first** Select initial periods of time series based on a date offset

**last** Select final periods of time series based on a date offset

**DatetimeIndex.indexer\_between\_time** Get just the index locations for values between particular times of the day

**Examples**

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09 00:00:00	1
2018-04-10 00:20:00	2
2018-04-11 00:40:00	3
2018-04-12 01:00:00	4

```
>>> ts.between_time('0:15', '0:45')
```

	A
2018-04-10 00:20:00	2
2018-04-11 00:40:00	3

You get the times that are *not* between two times by setting start\_time later than end\_time:

```
>>> ts.between_time('0:45', '0:15')
```

	A
2018-04-09 00:00:00	1
2018-04-12 01:00:00	4

**pandas.Series.bfill**

**Series.bfill** (axis=None, inplace=False, limit=None, downcast=None)  
 Synonym for *DataFrame.fillna* (method='bfill')

**pandas.Series.bool**

**Series.bool** ()  
 Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a ValueError if the PandasObject does not have exactly 1 element, or that element is not boolean

## pandas.Series.cat

`Series.cat()`

Accessor object for categorical properties of the Series values.

Be aware that assigning to *categories* is an inplace operation, while all methods return new categorical data per default (but can be called with *inplace=True*).

### Parameters

**data** [Series or CategoricalIndex]

### Examples

```
>>> s.cat.categories
>>> s.cat.categories = list('abc')
>>> s.cat.rename_categories(list('cab'))
>>> s.cat.reorder_categories(list('cab'))
>>> s.cat.add_categories(['d', 'e'])
>>> s.cat.remove_categories(['d'])
>>> s.cat.remove_unused_categories()
>>> s.cat.set_categories(list('abcde'))
>>> s.cat.as_ordered()
>>> s.cat.as_unordered()
```

## pandas.Series.clip

`Series.clip(lower=None, upper=None, axis=None, inplace=False, *args, **kwargs)`

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

**Parameters** **lower** : float or array\_like, default None

Minimum threshold value. All values below this threshold will be set to it.

**upper** : float or array\_like, default None

Maximum threshold value. All values above this threshold will be set to it.

**axis** : int or string axis name, optional

Align object with lower and upper along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data.

New in version 0.21.0.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns** Series or DataFrame

Same type as calling object with the values outside the clip boundaries replaced

See also:

`clip_lower` Clip values below specified threshold(s).

`clip_upper` Clip values above specified threshold(s).

### Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9    -2
1     -3    -7
2      0     6
3     -1     8
4      5    -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6    -2
1     -3    -4
2      0     6
3     -1     6
4      5    -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0      2
1     -4
2     -1
3      6
4      3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6     2
1     -3    -4
2      0     3
3      6     8
4      5     3
```

### pandas.Series.clip\_lower

`Series.clip_lower(threshold, axis=None, inplace=False)`

Return copy of the input with values below a threshold truncated.

**Parameters** `threshold` : numeric or array-like

Minimum value allowed. All values below threshold will be set to this value.

- **float** : every value is compared to *threshold*.
- **array-like** : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for *axis=None*, or 1-D and the same length as the axis being compared.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Align *self* with *threshold* along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data.

New in version 0.21.0.

### Returns

**clipped** [same type as input]

### See also:

***Series.clip*** Return copy of input with values below and above thresholds truncated.

***Series.clip\_upper*** Return copy of input with values above threshold truncated.

### Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0      8
1      8
2      8
3      8
4      9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0      5
1      8
2      7
3      8
4      9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
```

(continues on next page)

(continued from previous page)

```
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

### pandas.Series.clip\_upper

`Series.clip_upper(threshold, axis=None, inplace=False)`

Return copy of input with values above given value(s) truncated.

#### Parameters

**threshold** [float or array\_like]

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

New in version 0.21.0.

#### Returns

**clipped** [same type as input]

See also:

`clip`

## pandas.Series.combine

`Series.combine` (*other*, *func*, *fill\_value*=nan)

Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other

### Parameters

**other** [Series or scalar value]

**func** : function

Function that takes two scalars as inputs and return a scalar

**fill\_value** [scalar value]

### Returns

**result** [Series]

See also:

[`Series.combine\_first`](#) Combine Series values, choosing the calling Series's values first

## Examples

```
>>> s1 = Series([1, 2])
>>> s2 = Series([0, 3])
>>> s1.combine(s2, lambda x1, x2: x1 if x1 < x2 else x2)
0    0
1    2
dtype: int64
```

## pandas.Series.combine\_first

`Series.combine_first` (*other*)

Combine Series values, choosing the calling Series's values first. Result index will be the union of the two indexes

### Parameters

**other** [Series]

### Returns

**combined** [Series]

See also:

[`Series.combine`](#) Perform elementwise operation on two Series using a given function

## Examples



```

>>> s1 = pd.Series([1, np.nan])
>>> s2 = pd.Series([3, 4])
>>> s1.combine_first(s2)
0    1.0
1    4.0
dtype: float64

```

### pandas.Series.compound

**Series.compound** (*axis=None, skipna=None, level=None*)

Return the compound percentage of the values for the requested axis

#### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**compounded** [scalar or Series (if level specified)]

### pandas.Series.compress

**Series.compress** (*condition, \*args, \*\*kwargs*)

Return selected slices of an array along given axis as a Series

#### See also:

`numpy.ndarray.compress`

### pandas.Series consolidate

**Series consolidate** (*inplace=False*)

Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

### pandas.Series.convert\_objects

**Series.convert\_objects** (*convert\_dates=True, convert\_numeric=False, convert\_timedeltas=True, copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

**Parameters** `convert_dates` : boolean, default True

If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

**convert\_numeric** : boolean, default False

If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

**convert\_timedeltas** : boolean, default True

If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

**copy** : boolean, default True

If True, return a copy even if no copy is necessary (e.g. no conversion was done). Note: This is meant for internal use, and should not be confused with inplace.

### Returns

**converted** [same as input object]

**See also:**

[`pandas.to\_datetime`](#) Convert argument to datetime.

[`pandas.to\_timedelta`](#) Convert argument to timedelta.

[`pandas.to\_numeric`](#) Return a fixed frequency timedelta index, with day as the default.

## pandas.Series.copy

`Series.copy (deep=True)`

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

**Parameters** `deep` : bool, default True

Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

**Returns** `copy` : Series, DataFrame or Panel

Object type matches caller.

### Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to `copy.deepcopy` in the Standard Library, which recursively copies object data (see examples below).

While Index objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since Index is immutable, the underlying data can be safely shared and a copy is not needed.

### Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

#### Shallow copy versus default (deep) copy:

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0      [10, 2]
1      [3, 4]
dtype: object
>>> deep
0      [10, 2]
1      [3, 4]
dtype: object
```

### **pandas.Series.corr**

`Series.corr` (*other*, *method*='pearson', *min\_periods*=None)  
Compute correlation with *other* Series, excluding missing values

#### **Parameters**

**other** [Series]

**method** : {'pearson', 'kendall', 'spearman'}

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

**min\_periods** : int, optional

Minimum number of observations needed to have a valid result

#### **Returns**

**correlation** [float]

### **pandas.Series.count**

`Series.count` (*level*=None)  
Return number of non-NA/null observations in the Series

**Parameters** **level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

#### **Returns**

**nobs** [int or Series (if level specified)]

### **pandas.Series.cov**

`Series.cov` (*other*, *min\_periods*=None)  
Compute covariance with Series, excluding missing values

**Parameters****other** [Series]**min\_periods** : int, optional

Minimum number of observations needed to have a valid result

**Returns****covariance** [float]**Normalized by N-1 (unbiased estimator).****pandas.Series.cummax**`Series.cummax(axis=None, skipna=True, *args, **kwargs)`

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

**Parameters axis** : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs** :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns****cummax** [scalar or Series]**See also:****`pandas.core.window.Expanding.max`** Similar functionality but ignores NaN values.**`Series.max`** Return the maximum over Series axis.**`Series.cummax`** Return cumulative maximum over Series axis.**`Series.cummin`** Return cumulative minimum over Series axis.**`Series.cumsum`** Return cumulative sum over Series axis.**`Series.cumprod`** Return cumulative product over Series axis.**Examples****Series**

```

>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0

```

(continues on next page)

(continued from previous page)

```
4      0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0      2.0
1      NaN
2      5.0
3      5.0
4      5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0      2.0
1      NaN
2      NaN
3      NaN
4      NaN
dtype: float64
```

## DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
   A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
   A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

**pandas.Series.cummin**

`Series.cummin` (*axis=None*, *skipna=True*, *\*args*, *\*\*kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

**Parameters** *axis* : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs** :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

**cummin** [scalar or Series]

**See also:**

[`pandas.core.window.Expanding.min`](#) Similar functionality but ignores NaN values.

[`Series.min`](#) Return the minimum over Series axis.

[`Series.cummax`](#) Return cumulative maximum over Series axis.

[`Series.cummin`](#) Return cumulative minimum over Series axis.

[`Series.cumsum`](#) Return cumulative sum over Series axis.

[`Series.cumprod`](#) Return cumulative product over Series axis.

**Examples****Series**

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0      2.0
1      NaN
2      NaN
3      NaN
4      NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
   A    B
0  2.0  1.0
1  2.0  NaN
2  1.0  0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

## pandas.Series.cumprod

`Series.cumprod(axis=None, skipna=True, *args, **kwargs)`

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

**Parameters** `axis`: {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs**:

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**



**cumprod** [scalar or Series]

See also:

**pandas.core.window.Expanding.prod** Similar functionality but ignores NaN values.

**Series.prod** Return the product over Series axis.

**Series.cummax** Return cumulative maximum over Series axis.

**Series.cummin** Return cumulative minimum over Series axis.

**Series.cumsum** Return cumulative sum over Series axis.

**Series.cumprod** Return cumulative product over Series axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0      2.0
1      NaN
2      5.0
3     -1.0
4      0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0      2.0
1      NaN
2     10.0
3    -10.0
4     -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0      2.0
1      NaN
2      NaN
3      NaN
4      NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0   0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
      A      B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
      A      B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

## pandas.Series.cumsum

`Series.cumsum(axis=None, skipna=True, *args, **kwargs)`

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

**Parameters** `axis` : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs** :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**cumsum** [scalar or Series]

**See also:**

[`pandas.core.window.Expanding.sum`](#) Similar functionality but ignores NaN values.

[`Series.sum`](#) Return the sum over Series axis.

[`Series.cummax`](#) Return cumulative maximum over Series axis.

[`Series.cummin`](#) Return cumulative minimum over Series axis.

[`Series.cumsum`](#) Return cumulative sum over Series axis.

[`Series.cumprod`](#) Return cumulative product over Series axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
   A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
   A    B
0  2.0  3.0
1  3.0 NaN
2  1.0  1.0
```

## pandas.Series.describe

`Series.describe` (*percentiles=None, include=None, exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters** `percentiles` : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

**include** : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use 'category'
- None (default) : The result will include all numeric columns.

**exclude** : list-like of dtypes or None (default), optional,

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(exclude=['O'])`). To exclude pandas categorical columns, use 'category'
- None (default) : The result will exclude nothing.

## Returns

**summary:** Series/DataFrame of summary statistics

## See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

## Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as `lower`, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

## Examples

Describing a numeric `Series`.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a categorical `Series`.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
top        a
freq       2
dtype: object
```

Describing a timestamp `Series`.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count      3
unique     2
top        2010-01-01 00:00:00
```

(continues on next page)

(continued from previous page)

```
freq                2
first      2000-01-01 00:00:00
last       2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                      'numeric': [1, 2, 3],
...                      'categorical': pd.Categorical(['d', 'e', 'f'])
...                      })
>>> df.describe()
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a DataFrame regardless of data type.

```
>>> df.describe(include='all')
      categorical  numeric object
count           3      3.0      3
unique          3      NaN      3
top            f      NaN      c
freq           1      NaN      1
mean          NaN      2.0     NaN
std           NaN      1.0     NaN
min           NaN      1.0     NaN
25%           NaN      1.5     NaN
50%           NaN      2.0     NaN
75%           NaN      2.5     NaN
max           NaN      3.0     NaN
```

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```
>>> df.describe(include=[np.number])
      numeric
count      3.0
```

(continues on next page)

(continued from previous page)

mean	2.0
std	1.0
min	1.0
25%	1.5
50%	2.0
75%	2.5
max	3.0

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
      object
count      3
unique      3
top         c
freq        1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
      categorical
count           3
unique           3
top             f
freq            1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
      categorical  object
count           3      3
unique           3      3
top             f      c
freq            1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
      categorical  numeric
count           3      3.0
unique           3      NaN
top             f      NaN
freq            1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

## pandas.Series.diff

Series.**diff** (*periods=1*)

First discrete difference of element.

Calculates the difference of a Series element compared with another element in the Series (default is element in previous row).

**Parameters** `periods` : int, default 1

Periods to shift for calculating difference, accepts negative values.

**Returns**

**diffed** [Series]

**See also:**

***Series.pct\_change*** Percent change over given number of periods.

***Series.shift*** Shift index by desired number of periods with an optional time freq.

***DataFrame.diff*** First discrete difference of object

## Examples

Difference with previous row

```
>>> s = pd.Series([1, 1, 2, 3, 5, 8])
>>> s.diff()
0    NaN
1    0.0
2    1.0
3    1.0
4    2.0
5    3.0
dtype: float64
```

Difference with 3rd previous row

```
>>> s.diff(periods=3)
0    NaN
1    NaN
2    NaN
3    2.0
4    4.0
5    6.0
dtype: float64
```

Difference with following row

```
>>> s.diff(periods=-1)
0    0.0
1   -1.0
2   -1.0
3   -2.0
4   -3.0
5    NaN
dtype: float64
```



### pandas.Series.div

`Series.div` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

[`Series.rtruediv`](#)

### Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

### pandas.Series.divide

`Series.divide` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

`Series.rtruediv`

#### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

### pandas.Series.divmod

`Series.divmod(other, level=None, fill_value=None, axis=0)`

Integer division and modulo of series and other, element-wise (binary operator *divmod*).

Equivalent to `series divmod other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

Series.None

### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.dot

Series.**dot** (*other*)

Matrix multiplication with DataFrame or inner-product with Series objects. Can also be called using *self* @ *other* in Python >= 3.5.

### Parameters

**other** [Series or DataFrame]

### Returns

**dot\_product** [scalar or Series]

## pandas.Series.drop

`Series.drop` (*labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise'*)  
Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels. When using a multi-index, labels on different levels can be removed by specifying the level.

**Parameters** **labels** : single label or list-like

Index labels to drop.

**axis** : 0, default 0

Redundant for application on Series.

**index, columns** : None

Redundant for application on Series, but index can be used instead of labels.

New in version 0.21.0.

**level** : int or level name, optional

For MultiIndex, level for which the labels will be removed.

**inplace** : bool, default False

If True, do operation inplace and return None.

**errors** : { 'ignore', 'raise' }, default 'raise'

If 'ignore', suppress error and only existing labels are dropped.

### Returns

**dropped** [pandas.Series]

### Raises

**KeyError**

If none of the labels are found in the index.

### See also:

[\*Series.reindex\*](#) Return only specified index labels of Series.

[\*Series.dropna\*](#) Return series without null values.

[\*Series.drop\\_duplicates\*](#) Return Series with duplicate values removed.

[\*DataFrame.drop\*](#) Drop specified labels from rows or columns.

## Examples

```
>>> s = pd.Series(data=np.arange(3), index=['A', 'B', 'C'])
>>> s
A    0
B    1
C    2
dtype: int64
```

Drop labels B en C

```
>>> s.drop(labels=['B', 'C'])
A    0
dtype: int64
```

Drop 2nd level label in MultiIndex Series

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                             ['speed', 'weight', 'length']],
...                       labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                              [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> s = pd.Series([45, 200, 1.2, 30, 250, 1.5, 320, 1, 0.3],
...               index=midx)
>>> s
lama      speed      45.0
          weight     200.0
          length      1.2
cow       speed      30.0
          weight     250.0
          length      1.5
falcon    speed     320.0
          weight      1.0
          length      0.3
dtype: float64
```

```
>>> s.drop(labels='weight', level=1)
lama      speed      45.0
          length      1.2
cow       speed      30.0
          length      1.5
falcon    speed     320.0
          length      0.3
dtype: float64
```

## pandas.Series.drop\_duplicates

**Series.drop\_duplicates** (*keep='first', inplace=False*)

Return Series with duplicate values removed.

**Parameters** **keep** : {'first', 'last', False}, default 'first'

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

**inplace** : boolean, default False

If True, performs operation inplace and returns None.

**Returns**

**deduplicated** [Series]

**See also:**

**Index.drop\_duplicates** equivalent method on Index

**DataFrame.drop\_duplicates** equivalent method on DataFrame

`Series.duplicated` related method on Series, indicating duplicate Series values.

## Examples

Generate an Series with duplicated entries.

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
...               name='animal')
>>> s
0      lama
1       cow
2      lama
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

With the ‘keep’ parameter, the selection behaviour of duplicated values can be changed. The value ‘first’ keeps the first occurrence for each set of duplicated entries. The default value of keep is ‘first’.

```
>>> s.drop_duplicates()
0      lama
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

The value ‘last’ for parameter ‘keep’ keeps the last occurrence for each set of duplicated entries.

```
>>> s.drop_duplicates(keep='last')
1       cow
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

The value `False` for parameter ‘keep’ discards all sets of duplicated entries. Setting the value of ‘inplace’ to `True` performs the operation inplace and returns `None`.

```
>>> s.drop_duplicates(keep=False, inplace=True)
>>> s
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

## pandas.Series.dropna

`Series.dropna (axis=0, inplace=False, **kwargs)`

Return a new Series with missing values removed.

See the [User Guide](#) for more on which values are considered missing, and how to work with missing data.

**Parameters** `axis`: {0 or ‘index’}, default 0

There is only one axis to drop values from.

**inplace** : bool, default False

If True, do operation inplace and return None.

**\*\*kwargs**

Not in use.

### Returns Series

Series with NA entries dropped from it.

### See also:

**Series.isna** Indicate missing values.

**Series.notna** Indicate existing (non-missing) values.

**Series.fillna** Replace missing values.

**DataFrame.dropna** Drop rows or columns which contain NA values.

**Index.dropna** Drop missing indices.

### Examples

```
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2     NaN
dtype: float64
```

Drop NA values from a Series.

```
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```

Keep the Series with valid entries in the same variable.

```
>>> ser.dropna(inplace=True)
>>> ser
0    1.0
1    2.0
dtype: float64
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])
>>> ser
0     NaN
1         2
2     NaT
3
4     None
5    I stay
dtype: object
```

(continues on next page)

(continued from previous page)

```
>>> ser.dropna()
1      2
3
5      I stay
dtype: object
```

## pandas.Series.dt

`Series.dt()`

Accessor object for datetimelike properties of the Series values.

### Examples

```
>>> s.dt.hour
>>> s.dt.second
>>> s.dt.quarter
```

Returns a Series indexed like the original Series. Raises `TypeError` if the Series does not contain datetime-like values.

## pandas.Series.duplicated

`Series.duplicated(keep='first')`

Indicate duplicate Series values.

Duplicated values are indicated as `True` values in the resulting Series. Either all duplicates, all except the first or all except the last occurrence of duplicates can be indicated.

**Parameters** `keep` : { 'first', 'last', False }, default 'first'

- 'first' : Mark duplicates as `True` except for the first occurrence.
- 'last' : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

**Returns**

`pandas.core.series.Series`

**See also:**

[`pandas.Index.duplicated`](#) Equivalent method on `pandas.Index`

[`pandas.DataFrame.duplicated`](#) Equivalent method on `pandas.DataFrame`

[`pandas.Series.drop\_duplicates`](#) Remove duplicate values from Series

### Examples

By default, for each set of duplicated values, the first occurrence is set on `False` and all others on `True`:



```
>>> animals = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> animals.duplicated()
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

which is equivalent to

```
>>> animals.duplicated(keep='first')
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

By using 'last', the last occurrence of each set of duplicated values is set on False and all others on True:

```
>>> animals.duplicated(keep='last')
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

By setting keep on False, all duplicates are True:

```
>>> animals.duplicated(keep=False)
0     True
1    False
2     True
3    False
4     True
dtype: bool
```

## pandas.Series.eq

`Series.eq(other, level=None, fill_value=None, axis=0)`

Equal to of series and other, element-wise (binary operator *eq*).

Equivalent to `series == other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

`Series.None`

### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.equals

`Series.equals` (*other*)

Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

## pandas.Series.ewm

`Series.ewm` (*com=None, span=None, halflife=None, alpha=None, min\_periods=0, adjust=True, ignore\_na=False, axis=0*)

Provides exponential weighted functions

New in version 0.18.0.

**Parameters** *com* : float, optional

Specify decay in terms of center of mass,  $\alpha = 1/(1 + com)$ , for  $com \geq 0$

**span** : float, optional

Specify decay in terms of span,  $\alpha = 2/(span + 1)$ , for  $span \geq 1$

**halflife** : float, optional

Specify decay in terms of half-life,  $\alpha = 1 - \exp(\log(0.5)/half\_life)$ , for  $half\_life > 0$

**alpha** : float, optional

Specify smoothing factor  $\alpha$  directly,  $0 < \alpha \leq 1$

New in version 0.18.0.

**min\_periods** : int, default 0

Minimum number of observations in window required to have a value (otherwise result is NA).

**adjust** : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

**ignore\_na** : boolean, default False

Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

### Returns

**a Window sub-classed for the particular operation**

See also:

*rolling* Provides rolling window calculations

*expanding* Provides expanding transformations.

### Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights  $(1-\alpha)^{(n-1)}$ ,  $(1-\alpha)^{(n-2)}$ , ...,  $1-\alpha$ , 1.

**When adjust is False, weighted averages are calculated recursively as:**  $weighted\_average[0] = arg[0]$ ;  $weighted\_average[i] = (1-\alpha)*weighted\_average[i-1] + \alpha*arg[i]$ .

When ignore\_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are  $(1-\alpha)^2$  and 1 (if adjust is True), and  $(1-\alpha)^2$  and  $\alpha$  (if adjust is False).

When ignore\_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are  $1-\alpha$  and 1 (if adjust is True), and  $1-\alpha$  and  $\alpha$  (if adjust is False).

More details can be found at <http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows>

### Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
      B
0    0.0
1    1.0
2    2.0
3   NaN
4    4.0
```

```
>>> df.ewm(com=0.5).mean()
      B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

## pandas.Series.expanding

`Series.expanding` (*min\_periods=1, center=False, axis=0*)

Provides expanding transformations.

New in version 0.18.0.

**Parameters** `min_periods` : int, default 1

Minimum number of observations in window required to have a value (otherwise result is NA).

**center** : boolean, default False

Set the labels at the center of the window.

**axis** [int or string, default 0]

### Returns

a **Window** sub-classed for the particular operation

**See also:**

[\*rolling\*](#) Provides rolling window calculations

[\*ewm\*](#) Provides exponential weighted functions

### Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

### Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
      B
0    0.0
1    1.0
2    2.0
3   NaN
4    4.0
```

```
>>> df.expanding(2).sum()
      B
0   NaN
1    1.0
2    3.0
3    3.0
4    7.0
```

## pandas.Series.factorize

`Series.factorize` (*sort=False*, *na\_sentinel=-1*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

**Parameters** *sort* : boolean, default False

Sort *uniques* and shuffle *labels* to maintain the relationship.

**na\_sentinel** : int, default -1

Value to mark “not found”.

**Returns** *labels* : ndarray

An integer ndarray that’s an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

**uniques** : ndarray, Index, or Categorical

The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

---

**Note:** Even if there’s a missing value in *values*, *uniques* will *not* contain an entry for it.

---

**See also:**

`pandas.cut` Discretize continuous-valued array.

`pandas.unique` Find the unique valuse in an array.

## Examples

These examples all show *factorize* as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With `sort=True`, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```

Missing values are indicated in *labels* with *na\_sentinel* (-1 by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that 'b' is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

## pandas.Series.ffill

`Series.fffll` (*axis=None, inplace=False, limit=None, downcast=None*)  
 Synonym for `DataFrame.fillna(method='ffill')`

**pandas.Series.fillna**

`Series.fillna` (*value=None, method=None, axis=None, inplace=False, limit=None, down-  
cast=None, \*\*kwargs*)

Fill NA/NaN values using the specified method

**Parameters** **value** : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

**method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**axis** [{0 or 'index'}]

**inplace** : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

**limit** : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

**downcast** : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

**Returns**

**filled** [Series]

**See also:**

*interpolate* Fill NaN values using interpolation.

*reindex, asfreq*

**Examples**

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
   A    B    C    D
0 NaN  2.0 NaN   0
1 3.0  4.0 NaN   1
```

(continues on next page)

(continued from previous page)

2	NaN	NaN	NaN	5
3	NaN	3.0	NaN	4

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
   A    B    C    D
0  0.0  2.0  0.0  0
1  3.0  4.0  0.0  1
2  0.0  0.0  0.0  5
3  0.0  3.0  0.0  4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
   A    B    C    D
0  NaN  2.0  NaN  0
1  3.0  4.0  NaN  1
2  3.0  4.0  NaN  5
3  3.0  3.0  NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  2.0  1
2  0.0  1.0  2.0  5
3  0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  NaN  1
2  NaN  1.0  NaN  5
3  NaN  3.0  NaN  4
```

## pandas.Series.filter

**Series.filter** (*items=None, like=None, regex=None, axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

**Parameters** **items** : list-like

List of info axis to restrict to (must not all be present)

**like** : string

Keep info axis where “arg in col == True”

**regex** : string (regular expression)



Keep info axis with `re.search(regex, col) == True`

**axis** : int or string axis name

The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

### Returns

same type as input object

### See also:

`pandas.DataFrame.loc`

### Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

### Examples

```
>>> df
one  two  three
mouse    1    2    3
rabbit   4    5    6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse    1    3
rabbit   4    6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse    1    3
rabbit   4    6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit   4    5    6
```

## pandas.Series.first

`Series.first(offset)`

Convenience method for subsetting initial periods of time series data based on a date offset.

### Parameters

**offset** [string, DateOffset, dateutil.relativedelta]

### Returns

**subset** [type of caller]

**Raises** `TypeError`

If the index is not a `DatetimeIndex`

**See also:**

**`last`** Select final periods of time series based on a date offset

**`at_time`** Select values at a particular time of the day

**`between_time`** Select values between particular times of the day

**Examples**

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09	1
2018-04-11	2
2018-04-13	3
2018-04-15	4

Get the rows for the first 3 days:

```
>>> ts.first('3D')
```

	A
2018-04-09	1
2018-04-11	2

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

**pandas.Series.first\_valid\_index**

`Series.first_valid_index()`  
Return index for first non-NA/null value.

**Returns**

**scalar** [type of index]

**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

**pandas.Series.floordiv**

`Series.floordiv(other, level=None, fill_value=None, axis=0)`  
Integer division of series and other, element-wise (binary operator *floordiv*).

Equivalent to `series // other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

**Parameters****other** [Series or scalar value]**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns****result** [Series]**See also:***Series.rfloordiv***Examples**

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

**pandas.Series.from\_array**

**classmethod** `Series.from_array(arr, index=None, name=None, dtype=None, copy=False, fastpath=False)`

Construct Series from array.

Deprecated since version 0.23.0: Use `pd.Series(..)` constructor instead.

## `pandas.Series.from_csv`

**classmethod** `Series.from_csv` (*path*, *sep*=' ', *parse\_dates*=True, *header*=None, *index\_col*=0, *encoding*=None, *infer\_datetime\_format*=False)

Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a time Series.

This method only differs from `pandas.read_csv()` in some defaults:

- *index\_col* is 0 instead of None (take first column as index by default)
- *header* is None instead of 0 (the first row is not used as the column names)
- *parse\_dates* is True instead of False (try parsing the index as datetime by default)

With `pandas.read_csv()`, the option `squeeze=True` can be used to return a Series like `from_csv`.

### Parameters

**path** [string file path or file handle / StringIO]

**sep** : string, default ' '

Field delimiter

**parse\_dates** : boolean, default True

Parse dates. Different default from `read_table`

**header** : int, default None

Row to use as header (skip prior rows)

**index\_col** : int or sequence, default 0

Column to use for index. If a sequence is given, a MultiIndex is used. Different default from `read_table`

**encoding** : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

**infer\_datetime\_format**: boolean, default False

If True and *parse\_dates* is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

### Returns

**y** [Series]

**See also:**

`pandas.read_csv`

## pandas.Series.ge

`Series.ge` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Greater than or equal to of series and other, element-wise (binary operator *ge*).

Equivalent to `series >= other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

`Series.None`

## Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

## pandas.Series.get

`Series.get` (*key*, *default=None*)

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not

found.

**Parameters**

**key** [object]

**Returns**

**value** [type of items contained in object]

**pandas.Series.get\_dtype\_counts**

`Series.get_dtype_counts()`

Return counts of unique dtypes in this object.

**Returns dtype** : Series

Series with the count of columns with each dtype.

**See also:**

**dtypes** Return the dtypes in this object.

**Examples**

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
   str  int  float
0   a    1   1.0
1   b    2   2.0
2   c    3   3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

**pandas.Series.get\_ftype\_counts**

`Series.get_ftype_counts()`

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

**Returns dtype** : Series

Series with the count of columns with each type and sparsity (dense/sparse)

**See also:**

**ftypes** Return ftypes (indication of sparse/dense and dtype) in this object.

## Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
   str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

## pandas.Series.get\_value

`Series.get_value(label, takeable=False)`

Quickly retrieve single value at passed index label

Deprecated since version 0.21.0: Please use `.at[]` or `.iat[]` accessors.

### Parameters

**label** [object]

**takeable** [interpret the index as indexers, default False]

### Returns

**value** [scalar value]

## pandas.Series.get\_values

`Series.get_values()`

same as `values` (but handles sparseness conversions); is a view

## pandas.Series.groupby

`Series.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True, squeeze=False, observed=False, **kwargs)`

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

**Parameters** **by** : mapping, function, label, or list of labels

Used to determine the groups for the `groupby`. If `by` is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in `self`. Notice that a tuple is interpreted a (single) key.

**axis** [int, default 0]

**level** : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

**as\_index** : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as\_index=False is effectively “SQL-style” grouped output

**sort** : boolean, default True

Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.

**group\_keys** : boolean, default True

When calling apply, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**observed** : boolean, default False

This only applies if any of the groupers are Categoricals If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

## Returns

### GroupBy object

See also:

[\*resample\*](#) Convenience method for frequency conversion and resampling of time series.

## Notes

See the [user guide](#) for more.

## Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

## pandas.Series.gt

`Series.gt` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Greater than of series and other, element-wise (binary operator *gt*).



Equivalent to `series > other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

`Series.None`

#### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

### pandas.Series.head

`Series.head(n=5)`

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

**Parameters** *n* : int, default 5

Number of rows to select.

**Returns** `obj_head` : type of caller

The first  $n$  rows of the caller object.

**See also:**

`pandas.DataFrame.tail` Returns the last  $n$  rows.

## Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0  alligator
1      bee
2   falcon
3     lion
4   monkey
5   parrot
6    shark
7   whale
8   zebra
```

Viewing the first 5 lines

```
>>> df.head()
   animal
0  alligator
1      bee
2   falcon
3     lion
4   monkey
```

Viewing the first  $n$  lines (three in this case)

```
>>> df.head(3)
   animal
0  alligator
1      bee
2   falcon
```

## pandas.Series.hist

`Series.hist` (`by=None`, `ax=None`, `grid=True`, `xlabelsize=None`, `xrot=None`, `ylabelsize=None`, `yrot=None`, `figsize=None`, `bins=10`, `**kwargs`)

Draw histogram of the input series using matplotlib

**Parameters** `by` : object, optional

If passed, then used to form histograms for separate groups

`ax` : matplotlib axis object

If not passed, uses `gca()`

**grid** : boolean, default True

Whether to show axis grid lines

**xlabelsize** : int, default None

If specified changes the x-axis label size

**xrot** : float, default None

rotation of x axis labels

**ylabelsize** : int, default None

If specified changes the y-axis label size

**yrot** : float, default None

rotation of y axis labels

**figsize** : tuple, default None

figure size in inches by default

**bins** : integer or sequence, default 10

Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

**bins: integer, default 10**

Number of histogram bins to be used

**\*\*kws** : keywords

To be passed to the actual plotting function

**See also:**

`matplotlib.axes.Axes.hist` Plot a histogram using matplotlib.

### pandas.Series.idxmax

`Series.idxmax` (*axis=0, skipna=True, \*args, \*\*kwargs*)

Return the row label of the maximum value.

If multiple values equal the maximum, the first row label with that value is returned.

**Parameters** **skipna** : boolean, default True

Exclude NA/null values. If the entire Series is NA, the result will be NA.

**axis** : int, default 0

For compatibility with DataFrame.idxmax. Redundant for application on Series.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

**idxmax** [Index of maximum of values.]

**Raises** **ValueError**

If the Series is empty.

#### See also:

`numpy.argmax` Return indices of the maximum values along the given axis.

`DataFrame.idxmax` Return index of first occurrence of maximum over requested axis.

`Series.idxmin` Return index *label* of the first occurrence of minimum of values.

#### Notes

This method is the Series version of `ndarray.argmax`. This method returns the label of the maximum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

#### Examples

```
>>> s = pd.Series(data=[1, None, 4, 3, 4],
...                 index=['A', 'B', 'C', 'D', 'E'])
>>> s
A    1.0
B    NaN
C    4.0
D    3.0
E    4.0
dtype: float64
```

```
>>> s.idxmax()
'C'
```

If `skipna` is False and there is an NA value in the data, the function returns `nan`.

```
>>> s.idxmax(skipna=False)
nan
```

### pandas.Series.idxmin

`Series.idxmin` (*axis=None*, *skipna=True*, *\*args*, *\*\*kwargs*)

Return the row label of the minimum value.

If multiple values equal the minimum, the first row label with that value is returned.

**Parameters** `skipna` : boolean, default True

Exclude NA/null values. If the entire Series is NA, the result will be NA.

**axis** : int, default 0

For compatibility with `DataFrame.idxmin`. Redundant for application on Series.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with NumPy.

#### Returns

**idxmin** [Index of minimum of values.]

**Raises ValueError**

If the Series is empty.

**See also:**

`numpy.argmax` Return indices of the minimum values along the given axis.

`DataFrame.idxmin` Return index of first occurrence of minimum over requested axis.

`Series.idxmax` Return index *label* of the first occurrence of maximum of values.

**Notes**

This method is the Series version of `ndarray.argmax`. This method returns the label of the minimum, while `ndarray.argmax` returns the position. To get the position, use `series.values.argmax()`.

**Examples**

```
>>> s = pd.Series(data=[1, None, 4, 1],
...                 index=['A', 'B', 'C', 'D'])
>>> s
A    1.0
B    NaN
C    4.0
D    1.0
dtype: float64
```

```
>>> s.idxmin()
'A'
```

If `skipna` is False and there is an NA value in the data, the function returns nan.

```
>>> s.idxmin(skipna=False)
nan
```

**pandas.Series.infer\_objects**

`Series.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

**Returns**

**converted** [same type as input object]

**See also:**

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to numeric typeR

## Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

## pandas.Series.interpolate

`Series.interpolate` (*method='linear', axis=0, limit=None, inplace=False, limit\_direction='forward', limit\_area=None, downcast=None, \*\*kwargs*)  
Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

**Parameters** `method`: {'linear', 'time', 'index', 'values', 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise\_polynomial', 'from\_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interpld`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the actual numerical values of the index.
- 'krogh', 'piecewise\_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the [scipy documentation](#) and [tutorial documentation](#)
- 'from\_derivatives' refers to `BPoly.from_derivatives` which replaces 'piecewise\_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from\_derivatives' which replaces 'piecewise\_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

**axis**: {0, 1}, default 0

- 0: fill column-by-column
- 1: fill row-by-row

**limit** : int, default None.

Maximum number of consecutive NaNs to fill. Must be greater than 0.

**limit\_direction** [{ 'forward', 'backward', 'both' }, default 'forward']

**limit\_area** : { 'inside', 'outside' }, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

**inplace** : bool, default False

Update the NDFrame in place if possible.

**downcast** : optional, 'infer' or None, defaults to None

Downcast dtypes if possible.

**kwargs** [keyword arguments to pass on to the interpolating function.]

### Returns

**Series or DataFrame of same shape interpolated at the NaNs**

**See also:**

[\*reindex\*](#), [\*replace\*](#), [\*fillna\*](#)

### Examples

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

## pandas.Series.isin

**Series.isin** (*values*)

Check whether *values* are contained in Series.

Return a boolean Series showing whether each element in the Series matches an element in the passed sequence of *values* exactly.

**Parameters** *values* : set or list-like

The sequence of values to test. Passing in a single string will raise a `TypeError`. Instead, turn a single string into a list of one element.

New in version 0.18.1: Support for values as a set.

### Returns

**isin** [Series (bool dtype)]

### Raises `TypeError`

- If *values* is a string

### See also:

`pandas.DataFrame.isin` equivalent method on `DataFrame`

### Examples

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama',
...               'hippo'], name='animal')
>>> s.isin(['cow', 'lama'])
0      True
1      True
2      True
3     False
4      True
5     False
Name: animal, dtype: bool
```

Passing a single string as `s.isin('lama')` will raise an error. Use a list of one element instead:

```
>>> s.isin(['lama'])
0      True
1     False
2      True
3     False
4      True
5     False
Name: animal, dtype: bool
```

## `pandas.Series.isna`

`Series.isna()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

### Returns `Series`

Mask of bool values for each element in `Series` that indicates whether an element is not an NA value.

### See also:



**Series.isnull** alias of isna

**Series.notna** boolean inverse of isna

**Series.dropna** omit axes labels with missing values

**isna** top-level isna

## Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born    name    toy
0  5.0      NaT  Alfred   None
1  6.0 1939-05-27  Batman Batmobile
2  NaN 1940-04-25         Joker
```

```
>>> df.isna()
   age  born  name  toy
0 False  True False  True
1 False False False False
2  True False False False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

## pandas.Series.isnull

**Series.isnull()**

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns Series**

Mask of bool values for each element in Series that indicates whether an element is not an NA value.

See also:

`Series.isnull` alias of `isna`

`Series.notna` boolean inverse of `isna`

`Series.dropna` omit axes labels with missing values

`isna` top-level `isna`

## Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born    name    toy
0  5.0      NaT  Alfred   None
1  6.0 1939-05-27  Batman Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.isna()
   age  born  name  toy
0 False  True False  True
1 False False False False
2  True False False False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

## pandas.Series.item

`Series.item()`

return the first element of the underlying data as a python scalar

**pandas.Series.items**

`Series.items()`  
 Lazily iterate over (index, value) tuples

**pandas.Series.iteritems**

`Series.iteritems()`  
 Lazily iterate over (index, value) tuples

**pandas.Series.keys**

`Series.keys()`  
 Alias for index

**pandas.Series.kurt**

`Series.kurt` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)  
 Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

**Parameters**

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**kurt** [scalar or Series (if level specified)]

**pandas.Series.kurtosis**

`Series.kurtosis` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)  
 Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

**Parameters**

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**kurt** [scalar or Series (if level specified)]

### pandas.Series.last

Series.**last** (*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

#### Parameters

**offset** [string, DateOffset, dateutil.relativedelta]

#### Returns

**subset** [type of caller]

#### Raises

**TypeError**

If the index is not a *DatetimeIndex*

#### See also:

**first** Select initial periods of time series based on a date offset

**at\_time** Select values at a particular time of the day

**between\_time** Select values between particular times of the day

### Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09	1
2018-04-11	2
2018-04-13	3
2018-04-15	4

Get the rows for the last 3 days:

```
>>> ts.last('3D')
```

	A
2018-04-13	3
2018-04-15	4

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

**pandas.Series.last\_valid\_index****Series.last\_valid\_index()**

Return index for last non-NA/null value.

**Returns****scalar** [type of index]**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

**pandas.Series.le****Series.le** (*other, level=None, fill\_value=None, axis=0*)Less than or equal to of series and other, element-wise (binary operator *le*).Equivalent to `series <= other`, but with support to substitute a `fill_value` for missing data in one of the inputs.**Parameters****other** [Series or scalar value]**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns****result** [Series]**See also:**

Series.None

**Examples**

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0

```

(continues on next page)

(continued from previous page)

```

e      NaN
dtype: float64
>>> a.add(b, fill_value=0)
a      2.0
b      1.0
c      1.0
d      1.0
e      NaN
dtype: float64

```

## pandas.Series.lt

`Series.lt` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Less than of series and other, element-wise (binary operator *lt*).

Equivalent to `series < other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

`Series.None`

## Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a      1.0
b      1.0
c      1.0
d      NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a      1.0
b      NaN
d      1.0
e      NaN
dtype: float64
>>> a.add(b, fill_value=0)

```

(continues on next page)

(continued from previous page)

```

a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

**pandas.Series.mad****Series.mad** (*axis=None, skipna=None, level=None*)

Return the mean absolute deviation of the values for the requested axis

**Parameters****axis** [{index (0)}]**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns****mad** [scalar or Series (if level specified)]**pandas.Series.map****Series.map** (*arg, na\_action=None*)

Map values of Series using input correspondence (a dict, Series, or function).

**Parameters** **arg** : function, dict, or Series

Mapping correspondence.

**na\_action** : {None, 'ignore'}

If 'ignore', propagate NA values, without passing them to the mapping correspondence.

**Returns** **y** : Series

Same index as caller.

**See also:****Series.apply** For applying more complex functions on a Series.**DataFrame.apply** Apply a function row-/column-wise.**DataFrame.applymap** Apply a function elementwise on a whole DataFrame.

## Notes

When *arg* is a dictionary, values in Series that are not in the dictionary (as keys) are converted to NaN. However, if the dictionary is a dict subclass that defines `__missing__` (i.e. provides a method for default values), then this default is used rather than NaN:

```
>>> from collections import Counter
>>> counter = Counter()
>>> counter['bar'] += 1
>>> y.map(counter)
1    0
2    1
3    0
dtype: int64
```

## Examples

Map inputs to outputs (both of type *Series*):

```
>>> x = pd.Series([1,2,3], index=['one', 'two', 'three'])
>>> x
one      1
two      2
three    3
dtype: int64
```

```
>>> y = pd.Series(['foo', 'bar', 'baz'], index=[1,2,3])
>>> y
1    foo
2    bar
3    baz
```

```
>>> x.map(y)
one    foo
two    bar
three  baz
```

If *arg* is a dictionary, return a new Series with values converted according to the dictionary's mapping:

```
>>> z = {1: 'A', 2: 'B', 3: 'C'}
```

```
>>> x.map(z)
one    A
two    B
three  C
```

Use *na\_action* to control whether NA values are affected by the mapping function.

```
>>> s = pd.Series([1, 2, 3, np.nan])
```

```
>>> s2 = s.map('this is a string {}'.format, na_action=None)
0    this is a string 1.0
1    this is a string 2.0
2    this is a string 3.0
```

(continues on next page)



(continued from previous page)

```
3    this is a string nan
dtype: object
```

```
>>> s3 = s.map('this is a string {}'.format, na_action='ignore')
0    this is a string 1.0
1    this is a string 2.0
2    this is a string 3.0
3                                NaN
dtype: object
```

**pandas.Series.mask**

`Series.mask(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)`

Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

**Parameters** *cond* : boolean NDFrame, array-like, or callable

Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as *cond*.

**other** : scalar, NDFrame, or callable

Entries where *cond* is True are replaced with corresponding value from *other*. If *other* is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as *other*.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

**axis** [alignment axis if needed, default None]

**level** [alignment level if needed, default None]

**errors** : str, {'raise', 'ignore'}, default 'raise'

- *raise* : allow exceptions to be raised
- *ignore* : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

**Returns****wh** [same type as caller]**See also:**`DataFrame.where()`**Notes**

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in [indexing](#).

**Examples**

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A      B
0  True  True
```

(continues on next page)

(continued from previous page)

```

1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True

```

### pandas.Series.max

Series.**max** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

**This method returns the maximum of the values in the object.** If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

#### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**max** [scalar or Series (if level specified)]

### pandas.Series.mean

Series.**mean** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return the mean of the values for the requested axis

#### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**mean** [scalar or Series (if level specified)]

**pandas.Series.median**

`Series.median` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return the median of the values for the requested axis

**Parameters**

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**median** [scalar or Series (if level specified)]

**pandas.Series.memory\_usage**

`Series.memory_usage` (*index=True, deep=False*)

Return the memory usage of the Series.

The memory usage can optionally include the contribution of the index and of elements of *object* dtype.

**Parameters** **index** : bool, default True

Specifies whether to include the memory usage of the Series index.

**deep** : bool, default False

If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned value.

**Returns** **int**

Bytes of memory consumed.

**See also:**

`numpy.ndarray.nbytes` Total bytes consumed by the elements of the array.

`DataFrame.memory_usage` Bytes consumed by a DataFrame.

## Examples

```
>>> s = pd.Series(range(3))
>>> s.memory_usage()
104
```

Not including the index gives the size of the rest of the data, which is necessarily smaller:

```
>>> s.memory_usage(index=False)
24
```

The memory footprint of *object* values is ignored by default:

```
>>> s = pd.Series(["a", "b"])
>>> s.values
array(['a', 'b'], dtype=object)
>>> s.memory_usage()
96
>>> s.memory_usage(deep=True)
212
```

## pandas.Series.min

`Series.min(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

**This method returns the minimum of the values in the object.** If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**min** [scalar or Series (if level specified)]

## pandas.Series.mod

`Series.mod(other, level=None, fill_value=None, axis=0)`

Modulo of series and other, element-wise (binary operator `mod`).

Equivalent to `series % other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

**Parameters**

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

**result** [Series]

**See also:**

*Series.rmod*

**Examples**

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

**pandas.Series.mode**

`Series.mode()`

Return the mode(s) of the dataset.

Always returns Series even if only one value is returned.

**Returns**

**modes** [Series (sorted)]

**pandas.Series.mul**

`Series.mul` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

**Parameters**

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

**result** [Series]

**See also:**

[`Series.rmul`](#)

**Examples**

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

**pandas.Series.multiply**

`Series.multiply` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*).

Equivalent to `series * other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

`Series.rmul`

#### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

### pandas.Series.ne

`Series.ne` (*other, level=None, fill\_value=None, axis=0*)

Not equal to of series and other, element-wise (binary operator *ne*).

Equivalent to `series != other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters



**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

Series.None

### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.nlargest

Series.**nlargest** (*n=5, keep='first'*)

Return the largest *n* elements.

**Parameters** *n* : int

Return this many descending sorted values

**keep** : {'first', 'last'}, default 'first'

Where there are duplicate values: - *first* : take the first occurrence. - *last* : take the last occurrence.

**Returns** *top\_n* : Series

The  $n$  largest values in the Series, in sorted order

**See also:**

`Series.nsmallest`

## Notes

Faster than `.sort_values(ascending=False).head(n)` for small  $n$  relative to the size of the Series object.

## Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nlargest(10)  # only sorts up to the N requested
219921    4.644710
82124     4.608745
421689    4.564644
425277    4.447014
718691    4.414137
43154     4.403520
283187    4.313922
595519    4.273635
503969    4.250236
121637    4.240952
dtype: float64
```

## pandas.Series.nonzero

`Series.nonzero()`

Return the *integer* indices of the elements that are non-zero

This method is equivalent to calling `numpy.nonzero` on the series data. For compatibility with NumPy, the return value is the same (a tuple with an array of indices for each dimension), but it will always be a one-item tuple because series only have one dimension.

**See also:**

`numpy.nonzero`

## Examples

```
>>> s = pd.Series([0, 3, 0, 4])
>>> s.nonzero()
(array([1, 3]),)
>>> s.iloc[s.nonzero()[0]]
1    3
3    4
dtype: int64
```

```
>>> s = pd.Series([0, 3, 0, 4], index=['a', 'b', 'c', 'd'])
# same return although index of s is different
>>> s.nonzero()
(array([1, 3]),)
>>> s.iloc[s.nonzero()[0]]
b      3
d      4
dtype: int64
```

## pandas.Series.notna

`Series.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to False values.

### Returns Series

Mask of bool values for each element in Series that indicates whether an element is not an NA value.

### See also:

**`Series.notnull`** alias of `notna`

**`Series.isna`** boolean inverse of `notna`

**`Series.dropna`** omit axes labels with missing values

**`notna`** top-level `notna`

## Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born   name      toy
0  5.0      NaT  Alfred     None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notna()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

## pandas.Series.notnull

`Series.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to False values.

### Returns Series

Mask of bool values for each element in Series that indicates whether an element is not an NA value.

### See also:

[`Series.notnull`](#) alias of `notna`

[`Series.isna`](#) boolean inverse of `notna`

[`Series.dropna`](#) omit axes labels with missing values

[`notna`](#) top-level `notna`

## Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born      name      toy
0  5.0        NaT  Alfred    None
1  6.0  1939-05-27  Batman  Batmobile
2  NaN  1940-04-25      Joker
```

```
>>> df.notna()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

## pandas.Series.nsmallest

`Series.nsmallest` ( $n=5$ , *keep*='first')

Return the smallest  $n$  elements.

**Parameters**  $n$  : int

Return this many ascending sorted values

**keep** : {'first', 'last'}, default 'first'

Where there are duplicate values: - *first* : take the first occurrence. - *last* : take the last occurrence.

**Returns** `bottom_n` : Series

The  $n$  smallest values in the Series, in sorted order

**See also:**

[`Series.nlargest`](#)

## Notes

Faster than `.sort_values().head(n)` for small  $n$  relative to the size of the Series object.

## Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nsmallest(10) # only sorts up to the N requested
288532    -4.954580
```

(continues on next page)

(continued from previous page)

```
732345    -4.835960
64803     -4.812550
446457    -4.609998
501225    -4.483945
669476    -4.472935
973615    -4.401699
621279    -4.355126
773916    -4.347355
359919    -4.331927
dtype: float64
```

## pandas.Series.nunique

`Series.nunique` (*dropna=True*)

Return number of unique elements in the object.

Excludes NA values by default.

**Parameters** `dropna` : boolean, default True

Don't include NaN in the count.

**Returns**

`nunique` [int]

## pandas.Series.pct\_change

`Series.pct_change` (*periods=1, fill\_method='pad', limit=None, freq=None, \*\*kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

**Parameters** `periods` : int, default 1

Periods to shift for forming percent change.

**fill\_method** : str, default 'pad'

How to handle NAs before computing percent changes.

**limit** : int, default None

The number of consecutive NAs to fill before stopping.

**freq** : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay()).

**\*\*kwargs**

Additional keyword arguments are passed into `DataFrame.shift` or `Series.shift`.

**Returns** `chg` : Series or DataFrame

The same type as the calling object.

**See also:**

[`Series.diff`](#) Compute the difference of two elements in a Series.

**DataFrame.diff** Compute the difference of two elements in a DataFrame.

**Series.shift** Shift the index by some number of periods.

**DataFrame.shift** Shift the index by some number of periods.

## Examples

### Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0      NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0      NaN
1      NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0      NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

### DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
```

(continues on next page)

(continued from previous page)

	FR	GR	IT
1980-01-01	4.0405	1.7246	804.74
1980-02-01	4.0963	1.7482	810.01
1980-03-01	4.3149	1.8519	860.13

```
>>> df.pct_change()
              FR          GR          IT
1980-01-01    NaN          NaN          NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
              2016          2015          2014
GOOG  1769950    1500923    1371819
APPL  30586265   40912316   41403351
```

```
>>> df.pct_change(axis='columns')
              2016          2015          2014
GOOG    NaN -0.151997 -0.086016
APPL    NaN  0.337604  0.012002
```

## pandas.Series.pipe

`Series.pipe(func, *args, **kwargs)`  
 Apply func(self, \*args, \*\*kwargs)

**Parameters** `func` : function

function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a (callable, `data_keyword`) tuple where `data_keyword` is a string indicating the keyword of callable that expects the NDFrame.

**args** : iterable, optional

positional arguments passed into `func`.

**kwargs** : mapping, optional

a dictionary of keyword arguments passed into `func`.

**Returns**

**object** [the return type of `func`.]

**See also:**

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`



## Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe(f, arg2=b, arg3=c)
...   )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe((f, 'arg2'), arg1=a, arg3=c)
...   )
```

## pandas.Series.plot

`Series.plot` (*kind='line', ax=None, figsize=None, use\_index=True, title=None, grid=None, legend=False, style=None, logx=False, logy=False, loglog=False, xticks=None, yticks=None, xlim=None, ylim=None, rot=None, fontsize=None, colormap=None, table=False, yerr=None, xerr=None, label=None, secondary\_y=False, \*\*kws*)

Make plots of Series using matplotlib / pylab.

*New in version 0.17.0:* Each plot kind has a corresponding method on the `Series.plot` accessor: `s.plot(kind='line')` is equivalent to `s.plot.line()`.

### Parameters

**data** [Series]

**kind** : str

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot

**ax** : matplotlib axes object

If not passed, uses `gca()`

**figsize** [a tuple (width, height) in inches]

**use\_index** : boolean, default True

Use index as ticks for x axis

**title** : string or list

Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and *subplots* is True, print each item in the list above the corresponding subplot.

**grid** : boolean, default None (matlab style default)

Axis grid lines

**legend** : False/True/'reverse'

Place legend on axis subplots

**style** : list or dict

matplotlib line style per column

**logx** : boolean, default False

Use log scaling on x axis

**logy** : boolean, default False

Use log scaling on y axis

**loglog** : boolean, default False

Use log scaling on both x and y axes

**xticks** : sequence

Values to use for the xticks

**yticks** : sequence

Values to use for the yticks

**xlim** [2-tuple/list]

**ylim** [2-tuple/list]

**rot** : int, default None

Rotation for ticks (xticks for vertical, yticks for horizontal plots)

**fontsize** : int, default None

Font size for xticks and yticks

**colormap** : str or matplotlib colormap object, default None

Colormap to select colors from. If string, load colormap with that name from matplotlib.

**colorbar** : boolean, optional

If True, plot colorbar (only relevant for 'scatter' and 'hexbin' plots)

**position** : float

Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

**table** : boolean, Series or DataFrame, default False

If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib's default layout. If a Series or DataFrame is passed, use passed data to draw a table.

**yerr** : DataFrame, Series, array-like, dict and str

See *Plotting with Error Bars* for detail.

**xerr** [same types as yerr.]

**label** [label argument to provide to plot]

**secondary\_y** : boolean or sequence of ints, default False

If True then y-axis will be on the right

**mark\_right** : boolean, default True

When using a secondary\_y axis, automatically mark the column labels with “(right)” in the legend

**\*\*kws** : keywords

Options to pass to matplotlib plotting method

#### Returns

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

#### Notes

- See matplotlib documentation online for more on this subject
- If *kind* = 'bar' or 'barh', you can specify relative alignments for bar plot layout by *position* keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

### pandas.Series.pop

`Series.pop(item)`

Return item and drop from frame. Raise KeyError if not found.

**Parameters** **item** : str

Column label to be popped

#### Returns

**popped** [Series]

#### Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                   columns=('name', 'class', 'max_speed'))
```

(continues on next page)

(continued from previous page)

```
>>> df
   name  class  max_speed
0  falcon   bird     389.0
1  parrot   bird      24.0
2   lion  mammal      80.5
3  monkey  mammal       NaN
```

```
>>> df.pop('class')
0    bird
1    bird
2  mammal
3  mammal
Name: class, dtype: object
```

```
>>> df
   name  max_speed
0  falcon     389.0
1  parrot     24.0
2   lion     80.5
3  monkey      NaN
```

## pandas.Series.pow

`Series.pow` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *pow*).

Equivalent to `series ** other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

`Series.rpow`

## Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
```

(continues on next page)

(continued from previous page)

```

b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

### pandas.Series.prod

`Series.prod` (*axis=None, skipna=None, level=None, numeric\_only=None, min\_count=0, \*\*kwargs*)

Return the product of the values for the requested axis

#### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

#### Returns

**prod** [scalar or Series (if level specified)]

### Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

## pandas.Series.product

`Series.product` (*axis=None*, *skipna=None*, *level=None*, *numeric\_only=None*, *min\_count=0*, *\*\*kwargs*)

Return the product of the values for the requested axis

### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

### Returns

**prod** [scalar or Series (if level specified)]

## Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

## pandas.Series.ptp

`Series.ptp` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

**Returns the difference between the maximum value and the** minimum value in the object. This is the equivalent of the `numpy.ndarray` method `ptp`.

### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**ptp** [scalar or Series (if level specified)]

## pandas.Series.put

`Series.put` (*\*args, \*\*kwargs*)

Applies the `put` method to its `values` attribute if it has one.

**See also:**

`numpy.ndarray.put`

## pandas.Series.quantile

`Series.quantile` (*q=0.5, interpolation='linear'*)

Return value at the given quantile, a la `numpy.percentile`.

**Parameters** **q** : float or array-like, default 0.5 (50% quantile)

$0 \leq q \leq 1$ , the quantile(s) to compute

**interpolation** : { 'linear', 'lower', 'higher', 'midpoint', 'nearest' }

New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points  $i$  and  $j$ :

- linear:  $i + (j - i) * fraction$ , where *fraction* is the fractional part of the index surrounded by  $i$  and  $j$ .
- lower:  $i$ .
- higher:  $j$ .
- nearest:  $i$  or  $j$  whichever is nearest.
- midpoint:  $(i + j) / 2$ .

**Returns** **quantile** : float or Series

if  $q$  is an array, a Series will be returned where the index is  $q$  and the values are the quantiles.

**See also:**

`pandas.core.window.Rolling.quantile`

## Examples

```
>>> s = Series([1, 2, 3, 4])
>>> s.quantile(.5)
2.5
>>> s.quantile([.25, .5, .75])
0.25    1.75
0.50    2.50
0.75    3.25
dtype: float64
```

## pandas.Series.radd

`Series.radd(other, level=None, fill_value=None, axis=0)`

Addition of series and other, element-wise (binary operator *radd*).

Equivalent to `other + series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level



**Returns****result** [Series]**See also:***Series.add***Examples**

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

**pandas.Series.rank**

`Series.rank` (*axis=0, method='average', numeric\_only=None, na\_option='keep', ascending=True, pct=False*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

index to direct ranking

**method** : {'average', 'min', 'max', 'first', 'dense'}

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**numeric\_only** : boolean, default None

Include only float, int, boolean data. Valid only for DataFrame or Panel objects

**na\_option** : {'keep', 'top', 'bottom'}

- **keep**: leave NA values where they are
- **top**: smallest rank if ascending
- **bottom**: smallest rank if descending

**ascending** : boolean, default True

False for ranks by high (1) to low (N)

**pct** : boolean, default False

Computes percentage rank of data

#### Returns

**ranks** [same type as caller]

### pandas.Series.ravel

`Series.ravel (order='C')`

Return the flattened underlying data as an ndarray

#### See also:

`numpy.ndarray.ravel`

### pandas.Series.rdiv

`Series.rdiv (other, level=None, fill_value=None, axis=0)`

Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

`Series.truediv`

### Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

### pandas.Series.reindex

**Series.reindex** (*index=None, \*\*kwargs*)

Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`

**Parameters** **index** : array-like, optional (should be specified using keywords)

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**method** : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

**copy** : boolean, default True

Return a new object, even if the passed indexes are the same

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value

**limit** : int, default None

Maximum number of consecutive elements to forward or backward fill

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation  $\text{abs}(\text{index}[\text{indexer}] - \text{target}) \leq \text{tolerance}$ .

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

## Returns

**reindexed** [Series]

## Examples

DataFrame.reindex supports two calling conventions

- (index=index\_labels, columns=column\_labels, ...)
- (labels, axis={'index', 'columns'}, ...)

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...     'http_status': [200, 200, 404, 404, 301],
...     'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...     index=index)
>>> df
```

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
```

	http_status	response_time
Safari	404.0	0.07
Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword `method` to fill the NaN values.

```
>>> df.reindex(new_index, fill_value=0)
           http_status  response_time
Safari              404             0.07
Iceweasel            0             0.00
Comodo Dragon        0             0.00
IE10                 404             0.08
Chrome               200             0.02
```

```
>>> df.reindex(new_index, fill_value='missing')
           http_status  response_time
Safari              404             0.07
Iceweasel          missing          missing
Comodo Dragon      missing          missing
IE10                 404             0.08
Chrome               200             0.02
```

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
           http_status  user_agent
Firefox              200         NaN
Chrome               200         NaN
Safari              404         NaN
IE10                 404         NaN
Konqueror            301         NaN
```

Or we can use “axis-style” keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
           http_status  user_agent
Firefox              200         NaN
Chrome               200         NaN
Safari              404         NaN
IE10                 404         NaN
Konqueror            301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                    index=date_index)
>>> df2
           prices
2010-01-01    100
2010-01-02    101
2010-01-03     NaN
2010-01-04    100
2010-01-05     89
2010-01-06     88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
           prices
2009-12-29     NaN
```

(continues on next page)

(continued from previous page)

2009-12-30	NaN
2009-12-31	NaN
2010-01-01	100
2010-01-02	101
2010-01-03	NaN
2010-01-04	100
2010-01-05	89
2010-01-06	88
2010-01-07	NaN

The index entries that did not have a value in the original data frame (for example, '2009-12-29') are by default filled with NaN. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the NaN values, pass `bfill` as an argument to the `method` keyword.

```
>>> df2.reindex(date_index2, method='bfill')
prices
2009-12-29    100
2009-12-30    100
2009-12-31    100
2010-01-01    100
2010-01-02    101
2010-01-03    NaN
2010-01-04    100
2010-01-05     89
2010-01-06     88
2010-01-07    NaN
```

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the `fillna()` method.

See the [user guide](#) for more.

## pandas.Series.reindex\_axis

`Series.reindex_axis(labels, axis=0, **kwargs)`

Conform Series to new index with optional filling logic.

Deprecated since version 0.21.0: Use `Series.reindex` instead.

## pandas.Series.reindex\_like

`Series.reindex_like(other, method=None, copy=True, limit=None, tolerance=None)`

Return an object with matching indices to myself.

### Parameters

**other** [Object]

**method** [string or None]

**copy** [boolean, default True]

**limit** : int, default None

Maximum number of consecutive labels to fill for inexact matches.

**tolerance** : optional

Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

### Returns

**reindexed** [same as input]

### Notes

Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`

## pandas.Series.rename

`Series.rename(index=None, **kwargs)`

Alter Series index labels or name

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

Alternatively, change `Series.name` with a scalar value.

See the [user guide](#) for more.

**Parameters** **index** : scalar, hashable sequence, dict-like or function, optional

dict-like or functions are transformations to apply to the index. Scalar or hashable sequence-like will alter the `Series.name` attribute.

**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False

Whether to return a new Series. If True then value of copy is ignored.

**level** : int or level name, default None

In case of a MultiIndex, only rename labels in the specified level.

### Returns

**renamed** [Series (new object)]

**See also:**

[pandas.Series.rename\\_axis](#)

### Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
```

(continues on next page)

(continued from previous page)

```

2      3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0      1
1      2
2      3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2) # function, changes labels
0      1
1      2
4      3
dtype: int64
>>> s.rename({1: 3, 2: 5}) # mapping, changes labels
0      1
3      2
5      3
dtype: int64

```

### pandas.Series.rename\_axis

**Series.rename\_axis** (*mapper*, *axis=0*, *copy=True*, *inplace=False*)  
 Alter the name of the index or columns.

**Parameters** **mapper** : scalar, list-like, optional

Value to set as the axis name attribute.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis.

**copy** : boolean, default True

Also copy underlying data.

**inplace** : boolean, default False

Modifies the object directly, instead of creating a new Series or DataFrame.

**Returns** **renamed** : Series, DataFrame, or None

The same type as the caller or None if *inplace* is True.

**See also:**

[\*pandas.Series.rename\*](#) Alter Series index labels or name

[\*pandas.DataFrame.rename\*](#) Alter DataFrame index labels or name

[\*pandas.Index.rename\*](#) Set new names on index

### Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.



## Examples

### Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

### DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
      A  B
foo
0     1  4
1     2  5
2     3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0     1  4
1     2  5
2     3  6
```

## pandas.Series.reorder\_levels

`Series.reorder_levels` (*order*)

Rearrange index levels using input order. May not drop or duplicate levels

**Parameters** `order` : list of int representing new level order.

(reference level by number or key)

**axis** [where to reorder levels]

**Returns**

type of caller (new object)

## pandas.Series.repeat

`Series.repeat` (*repeats*, *\*args*, *\*\*kwargs*)

Repeat elements of an Series. Refer to *numpy.ndarray.repeat* for more information about the *repeats* argument.

**See also:**

`numpy.ndarray.repeat`

## pandas.Series.replace

`Series.replace(to_replace=None, value=None, inplace=False, limit=None, regex=False, method='pad')`

Replace values given in *to\_replace* with *value*.

Values of the Series are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

**Parameters** *to\_replace* : str, regex, list, dict, Series, int, float, or None

How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to\_replace* will be replaced with *value*
  - str: string exactly matching *to\_replace* will be replaced with *value*
  - regex: regexs matching *to\_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to\_replace* and *value* are both lists, they **must** be the same length.
  - Second, if *regex=True* then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{ 'a': 'b', 'y': 'z' }` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{ 'a': 1, 'b': 'z' }` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be *None* in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{ 'a': { 'b': np.nan} }`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be *None* to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also *None* then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to replace any values matching *to\_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill.

**regex** : bool or same types as *to\_replace*, default False

Whether to interpret *to\_replace* and/or *value* as regular expressions. If this is True then *to\_replace* must be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to\_replace* must be None.

**method** : { 'pad', 'ffill', 'bfill', None }

The method to use when for replacement, when *to\_replace* is a scalar, list or tuple and *value* is None.

Changed in version 0.23.0: Added to DataFrame.

#### Returns Series

Object after replacement.

#### Raises AssertionError

- If *regex* is not a bool and *to\_replace* is not None.

#### TypeError

- If *to\_replace* is a dict and *value* is not a list, dict, ndarray, or Series
- If *to\_replace* is None and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple bool or datetime64 objects and the arguments to *to\_replace* does not match the type of the value being replaced

#### ValueError

- If a list or an ndarray is passed to *to\_replace* and *value* but they are not the same length.

#### See also:

[\*Series.fillna\*](#) Fill NA values

[\*Series.where\*](#) Replace values based on boolean condition

[\*Series.str.replace\*](#) Simple string replacement.

#### Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.

- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When dict is used as the *to\_replace* value, it is like key(s) in the dict are the *to\_replace* part and value(s) in the dict are the *value* parameter.

## Examples

### Scalar ‘to\_replace’ and ‘value’

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

### List-like ‘to\_replace’

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

**dict-like ‘to\_replace’**

```
>>> df.replace({0: 10, 1: 100})
   A  B  C
0  10  5  a
1 100  6  b
2   2  7  c
3   3  8  d
4   4  9  e
```

```
>>> df.replace({'A': 0, 'B': 5}, 100)
   A   B  C
0 100 100  a
1   1   6  b
2   2   7  c
3   3   8  d
4   4   9  e
```

```
>>> df.replace({'A': {0: 100, 4: 400}})
   A  B  C
0 100  5  a
1   1  6  b
2   2  7  c
3   3  8  d
4 400  9  e
```

**Regular expression ‘to\_replace’**

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
   A   B
0  new abc
1  foo new
2 bait xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
   A   B
0  new abc
1  foo bar
2 bait xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
   A   B
0  new abc
1  foo new
2 bait xyz
```

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
   A   B
0  new abc
1  xyz new
2 bait xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
   A   B
```

(continues on next page)

(continued from previous page)

```
0   new  abc
1   new  new
2  bait  xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the `to_replace` parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                    'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False}) # raises
Traceback (most recent call last):
...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the dict keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the `to_replace` parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the `to_replace` value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1     None
2     None
3         b
4     None
dtype: object
```

When `value=None` and `to_replace` is a scalar, list or tuple, `replace` uses the `method` parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0      10
1      10
2      10
3         b
4         b
dtype: object
```

## pandas.Series.resample

`Series.resample` (*rule*, *how=None*, *axis=0*, *fill\_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (`DatetimeIndex`, `PeriodIndex`, or `TimedeltaIndex`), or pass datetime-like values to the `on` or `level` keyword.

**Parameters** **rule** : string

the offset string or object representing target conversion

**axis** [int, optional, default 0]

**closed** : { 'right', 'left' }

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**label** : { 'right', 'left' }

Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**convention** : { 'start', 'end', 's', 'e' }

For PeriodIndex only, controls whether to use the start or end of *rule*

**kind**: { 'timestamp', 'period' }, optional

Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

**loffset** : timedelta

Adjust the resampled time labels

**base** : int, default 0

For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

**on** : string, optional

For a `DataFrame`, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

**level** : string or int, optional

For a `MultiIndex`, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

**Returns**

**Resampler object**

**See also:**

[\*groupby\*](#) Group by mapping, function, label, or list of labels.

## Notes

See the [user guide](#) for more.

To learn more about the offset strings, please see [this link](#).

## Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket 2000-01-01 00:03:00 contains the value 3, but the summed value in the resampled bucket with the label 2000-01-01 00:03:00 does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00    3
2000-01-01 00:06:00   12
2000-01-01 00:09:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00    0
2000-01-01 00:03:00    6
2000-01-01 00:06:00   15
2000-01-01 00:09:00   15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.



```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    1.0
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the NaN values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00     0
2000-01-01 00:00:30     0
2000-01-01 00:01:00     1
2000-01-01 00:01:30     1
2000-01-01 00:02:00     2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the NaN values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00     0
2000-01-01 00:00:30     1
2000-01-01 00:01:00     1
2000-01-01 00:01:30     2
2000-01-01 00:02:00     2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))

>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using 'start' *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
```

(continues on next page)

(continued from previous page)

```
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using ‘end’ *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
      a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                      columns=['a', 'b', 'c', 'd'],
                      index=pd.MultiIndex.from_product([time, [1, 2]]))
>>> df2.resample('3T', level=0).sum()
      a  b  c  d
2000-01-01 00:00:00  0  6 12 18
2000-01-01 00:03:00  0  4  8 12
```

## pandas.Series.reset\_index

`Series.reset_index (level=None, drop=False, name=None, inplace=False)`

Generate a new DataFrame or Series with the index reset.

This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

**Parameters** `level` : int, str, tuple, or list, default optional

For a Series with a MultiIndex, only remove the specified levels from the index.  
Removes all levels by default.

**drop** : bool, default False

Just reset the index, without inserting it as a column in the new DataFrame.

**name** : object, optional

The name to use for the column containing the original Series values. Uses `self.name` by default. This argument is ignored when `drop` is True.

**inplace** : bool, default False

Modify the Series in place (do not create a new object).

### Returns Series or DataFrame

When `drop` is False (the default), a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values. When `drop` is True, a *Series* is returned. In either case, if `inplace=True`, no value is returned.

### See also:

[`DataFrame.reset\_index`](#) Analogous function for DataFrame.

### Examples

```
>>> s = pd.Series([1, 2, 3, 4], name='foo',
...               index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```
>>> s.reset_index()
   idx  foo
0    a    1
1    b    2
2    c    3
3    d    4
```

To specify the name of the new column use `name`.

```
>>> s.reset_index(name='values')
   idx  values
0    a        1
1    b        2
2    c        3
3    d        4
```

To generate a new Series with the default set `drop` to True.

```
>>> s.reset_index(drop=True)
0    1
1    2
2    3
3    4
Name: foo, dtype: int64
```

To update the Series in place, without generating a new one set *inplace* to True. Note that it also requires *drop=True*.

```
>>> s.reset_index(inplace=True, drop=True)
>>> s
0    1
1    2
2    3
3    4
Name: foo, dtype: int64
```

The *level* parameter is interesting for Series with a multi-level index.

```
>>> arrays = [np.array(['bar', 'bar', 'baz', 'baz']),
...           np.array(['one', 'two', 'one', 'two'])]
>>> s2 = pd.Series(
...     range(4), name='foo',
...     index=pd.MultiIndex.from_arrays(arrays,
...                                     names=['a', 'b']))
```

To remove a specific level from the Index, use *level*.

```
>>> s2.reset_index(level='a')
      a  foo
b
one bar    0
two bar    1
one baz    2
two baz    3
```

If *level* is not set, all levels are removed from the Index.

```
>>> s2.reset_index()
      a  b  foo
0 bar one    0
1 bar two    1
2 baz one    2
3 baz two    3
```

## pandas.Series.rfloordiv

Series.**rfloordiv** (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // series`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

`Series.floordiv`

### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d     NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b     NaN
d    1.0
e     NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e     NaN
dtype: float64
```

## pandas.Series.rmod

`Series.rmod(other, level=None, fill_value=None, axis=0)`

Modulo of series and other, element-wise (binary operator *rmod*).

Equivalent to `other % series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

See also:

[`Series.mod`](#)

## Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.rmul

`Series.rmul` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *rmul*).

Equivalent to `other * series`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

See also:

[`Series.mul`](#)

## Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.rolling

`Series.rolling` (*window*, *min\_periods=None*, *center=False*, *win\_type=None*, *on=None*, *axis=0*, *closed=None*)

Provides rolling window calculations.

New in version 0.18.0.

**Parameters** *window* : int, or offset

Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

**min\_periods** : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

**center** : boolean, default False

Set the labels at the center of the window.

**win\_type** : string, default None

Provide a window type. If `None`, all points are evenly weighted. See the notes below for further information.

**on** : string, optional

For a DataFrame, column on which to calculate the rolling window, rather than the index

**closed** : string, default None

Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

**axis** [int or string, default 0]

### Returns

**a Window or Rolling sub-classed for the particular operation**

**See also:**

[\*expanding\*](#) Provides expanding transformations.

[\*ewm\*](#) Provides exponential weighted functions

### Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see [this link](#).

The recognized win\_types are:

- boxcar
- triang
- blackman
- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general\_gaussian (needs power, width)
- slepian (needs width).

If `win_type=None` all points are evenly weighted. To learn more about different window types see [scipy.signal window functions](#).



## Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
   B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, min\_periods defaults to the window length.

```
>>> df.rolling(2).sum()
   B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the min\_periods

```
>>> df.rolling(2, min_periods=1).sum()
   B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                     index = [pd.Timestamp('20130101 09:00:00'),
...                               pd.Timestamp('20130101 09:00:02'),
...                               pd.Timestamp('20130101 09:00:03'),
...                               pd.Timestamp('20130101 09:00:05'),
...                               pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for `min_periods` is 1.

```
>>> df.rolling('2s').sum()
      B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

## **pandas.Series.round**

`Series.round(decimals=0, *args, **kwargs)`

Round each value in a Series to the given number of decimals.

**Parameters** `decimals` : int

Number of decimal places to round to (default: 0). If `decimals` is negative, it specifies the number of positions to the left of the decimal point.

**Returns**

**Series object**

**See also:**

`numpy.around`, `DataFrame.round`

## **pandas.Series.rpow**

`Series.rpow(other, level=None, fill_value=None, axis=0)`

Exponential power of series and other, element-wise (binary operator *rpow*).

Equivalent to `other ** series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

**Parameters**

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

**result** [Series]

**See also:**

`Series.pow`

## Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.rsub

`Series.rsub` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *rsub*).

Equivalent to `other - series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

**See also:**

[`Series.sub`](#)

## Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

### pandas.Series.rtruediv

`Series.rtruediv` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / series`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

[`Series.truediv`](#)

### Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0

```

(continues on next page)

(continued from previous page)

```

b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64

```

### pandas.Series.sample

`Series.sample` (*n=None*, *frac=None*, *replace=False*, *weights=None*, *random\_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random\_state* for reproducibility.

**Parameters** *n* : int, optional

Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

**frac** : float, optional

Fraction of axis items to return. Cannot be used with *n*.

**replace** : boolean, optional

Sample with or without replacement. Default = False.

**weights** : str or ndarray-like, optional

Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when *axis* = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

**random\_state** : int or numpy.random.RandomState, optional

Seed for the random number generator (if int), or numpy RandomState object.

**axis** : int or string, optional

Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

## Returns

A new object of same type as caller.

## Examples

Generate an example Series and DataFrame:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0    -0.038497
1     1.820773
2    -0.972766
3    -1.598270
4    -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
      A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661
```

Next extract a random sample from both of these objects...

3 random elements from the Series:

```
>>> s.sample(n=3)
27    -0.994689
55    -1.049016
67    -0.224565
dtype: float64
```

And a random 10% of the DataFrame with replacement:

```
>>> df.sample(frac=0.1, replace=True)
      A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6   -0.148339  0.832938  1.787600 -1.383767
```

You can use *random state* for reproducibility:

```
>>> df.sample(random_state=1)
      A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264
```

**pandas.Series.searchsorted**`Series.searchsorted` (*value*, *side*='left', *sorter*=None)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted Series *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.**Parameters** *value* : array\_likeValues to insert into *self*.**side** : {'left', 'right'}, optionalIf 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).**sorter** : 1-D array\_like, optionalOptional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.**Returns** *indices* : array of intsArray of insertion points with the same shape as *value*.**See also:**`numpy.searchsorted`**Notes**

Binary search is used to find the required insertion points.

**Examples**

```
>>> x = pd.Series([1, 2, 3])
>>> x
0    1
1    2
2    3
dtype: int64
```

```
>>> x.searchsorted(4)
array([3])
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread',  
                        'cheese', 'milk'], ordered=True)  
[apple, bread, bread, cheese, milk]  
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')  
array([1])      # Note: an array, not a scalar
```

```
>>> x.searchsorted(['bread'], side='right')  
array([3])
```

## pandas.Series.select

`Series.select(crit, axis=0)`

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use `df.loc[df.index.map(crit)]` to select via labels

**Parameters** `crit` : function

To be called on each index (label). Should return True or False

**axis** [int]

**Returns**

**selection** [type of caller]

## pandas.Series.sem

`Series.sem(axis=None, skipna=None, level=None, ddof=1, numeric_only=None, **kwargs)`

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the `ddof` argument

**Parameters**

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**



**sem** [scalar or Series (if level specified)]

## pandas.Series.set\_axis

`Series.set_axis(labels, axis=0, inplace=None)`

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

**Parameters** **labels** : list-like, Index

The values for the new index.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

**inplace** : boolean, default None

Whether to return a new %(klass)s instance.

**Warning:** `inplace=None` currently falls back to `True`, but in a future version, will default to `False`. Use `inplace=True` explicitly rather than relying on the default.

**Returns** **renamed** : %(klass)s or None

An object of same type as caller if `inplace=False`, None otherwise.

**See also:**

[`pandas.DataFrame.rename\_axis`](#) Alter the name of the index or columns.

## Examples

### Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

### DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

## pandas.Series.set\_value

`Series.set_value (label, value, takeable=False)`

Quickly set single value at passed label. If label is not contained, a new object is created with the label placed at the end of the result index.

Deprecated since version 0.21.0: Please use `.at[]` or `.iat[]` accessors.

**Parameters** **label** : object

Partial indexing with MultiIndex not allowed

**value** : object

Scalar value

**takeable** [interpret the index as indexers, default False]

**Returns** **series** : Series

If label is contained, will be reference to calling Series, otherwise a new object

**pandas.Series.shift**`Series.shift` (*periods=1, freq=None, axis=0*)

Shift index by desired number of periods with an optional time freq

**Parameters** `periods` : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, optional

Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

**axis** [{0 or 'index'}]**Returns****shifted** [Series]**Notes**

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

**pandas.Series.skew**`Series.skew` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return unbiased skew over requested axis Normalized by N-1

**Parameters****axis** [{index (0)}]**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns****skew** [scalar or Series (if level specified)]**pandas.Series.slice\_shift**`Series.slice_shift` (*periods=1, axis=0*)

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

**Parameters** `periods` : int

Number of periods to move, can be positive or negative

#### Returns

**shifted** [same type as caller]

#### Notes

While the *slice\_shift* is faster than *shift*, you may pay for it later during alignment.

### pandas.Series.sort\_index

`Series.sort_index(axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na_position='last', sort_remaining=True)`

Sort Series by index labels.

Returns a new Series sorted by label if *inplace* argument is `False`, otherwise updates the original series and returns `None`.

**Parameters** **axis** : int, default 0

Axis to direct sorting. This can only be 0 for Series.

**level** : int, optional

If not `None`, sort on values in specified index level(s).

**ascending** : bool, default `true`

Sort ascending vs. descending.

**inplace** : bool, default `False`

If `True`, perform operation in-place.

**kind** : { 'quicksort', 'mergesort', 'heapsort' }, default 'quicksort'

Choice of sorting algorithm. See also `numpy.sort()` for more information. 'mergesort' is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

**na\_position** : { 'first', 'last' }, default 'last'

If 'first' puts NaNs at the beginning, 'last' puts NaNs at the end. Not implemented for MultiIndex.

**sort\_remaining** : bool, default `True`

If `true` and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level.

**Returns** **pandas.Series**

The original Series sorted by the labels

**See also:**

**DataFrame.sort\_index** Sort DataFrame by the index

**DataFrame.sort\_values** Sort DataFrame by the value

**Series.sort\_values** Sort Series by the value

## Examples

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, 4])
>>> s.sort_index()
1      c
2      b
3      a
4      d
dtype: object
```

### Sort Descending

```
>>> s.sort_index(ascending=False)
4      d
3      a
2      b
1      c
dtype: object
```

### Sort Inplace

```
>>> s.sort_index(inplace=True)
>>> s
1      c
2      b
3      a
4      d
dtype: object
```

By default NaNs are put at the end, but use *na\_position* to place them at the beginning

```
>>> s = pd.Series(['a', 'b', 'c', 'd'], index=[3, 2, 1, np.nan])
>>> s.sort_index(na_position='first')
NaN      d
1.0      c
2.0      b
3.0      a
dtype: object
```

### Specify index level to sort

```
>>> arrays = [np.array(['qux', 'qux', 'foo', 'foo',
...                     'baz', 'baz', 'bar', 'bar']),
...           np.array(['two', 'one', 'two', 'one',
...                     'two', 'one', 'two', 'one'])]
>>> s = pd.Series([1, 2, 3, 4, 5, 6, 7, 8], index=arrays)
>>> s.sort_index(level=1)
bar one      8
baz one      6
foo one      4
qux one      2
bar two      7
baz two      5
foo two      3
qux two      1
dtype: int64
```

Does not sort by remaining levels when sorting by levels

```
>>> s.sort_index(level=1, sort_remaining=False)
qux one 2
foo one 4
baz one 6
bar one 8
qux two 1
foo two 3
baz two 5
bar two 7
dtype: int64
```

## pandas.Series.sort\_values

`Series.sort_values` (*axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na\_position='last'*)

Sort by the values.

Sort a Series in ascending or descending order by some criterion.

**Parameters** *axis* : {0 or 'index'}, default 0

Axis to direct sorting. The value 'index' is accepted for compatibility with `DataFrame.sort_values`.

**ascending** : bool, default True

If True, sort values in ascending order, otherwise descending.

**inplace** : bool, default False

If True, perform operation in-place.

**kind** : {'quicksort', 'mergesort' or 'heapsort'}, default 'quicksort'

Choice of sorting algorithm. See also `numpy.sort()` for more information. 'mergesort' is the only stable algorithm.

**na\_position** : {'first' or 'last'}, default 'last'

Argument 'first' puts NaNs at the beginning, 'last' puts NaNs at the end.

**Returns** *Series*

Series ordered by values.

**See also:**

**`Series.sort_index`** Sort by the Series indices.

**`DataFrame.sort_values`** Sort DataFrame by the values along either axis.

**`DataFrame.sort_index`** Sort DataFrame by indices.

## Examples

```
>>> s = pd.Series([np.nan, 1, 3, 10, 5])
>>> s
0    NaN
1    1.0
```

(continues on next page)

(continued from previous page)

```
2      3.0
3     10.0
4      5.0
dtype: float64
```

Sort values ascending order (default behaviour)

```
>>> s.sort_values(ascending=True)
1      1.0
2      3.0
4      5.0
3     10.0
0      NaN
dtype: float64
```

Sort values descending order

```
>>> s.sort_values(ascending=False)
3     10.0
4      5.0
2      3.0
1      1.0
0      NaN
dtype: float64
```

Sort values inplace

```
>>> s.sort_values(ascending=False, inplace=True)
>>> s
3     10.0
4      5.0
2      3.0
1      1.0
0      NaN
dtype: float64
```

Sort values putting NAs first

```
>>> s.sort_values(na_position='first')
0      NaN
1      1.0
2      3.0
4      5.0
3     10.0
dtype: float64
```

Sort a series of strings

```
>>> s = pd.Series(['z', 'b', 'd', 'a', 'c'])
>>> s
0      z
1      b
2      d
3      a
4      c
dtype: object
```

```
>>> s.sort_values()
3      a
1      b
4      c
2      d
0      z
dtype: object
```

## pandas.Series.sortlevel

`Series.sortlevel` (*level=0, ascending=True, sort\_remaining=True*)

Sort Series with MultiIndex by chosen level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order),

Deprecated since version 0.20.0: Use `Series.sort_index()`

### Parameters

**level** [int or level name, default None]

**ascending** [bool, default True]

### Returns

**sorted** [Series]

### See also:

`Series.sort_index`

## pandas.Series.squeeze

`Series.squeeze` (*axis=None*)

Squeeze length 1 dimensions.

**Parameters** **axis** : None, integer or string axis name, optional

The axis to squeeze if 1-sized.

New in version 0.20.0.

### Returns

**scalar if 1-sized, else original object**

## pandas.Series.std

`Series.std` (*axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA



**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**std** [scalar or Series (if level specified)]

### pandas.Series.str

`Series.str()`

Vectorized string functions for Series and Index. NAs stay NA unless handled otherwise by a particular method. Patterned after Python's string methods, with some inspiration from R's stringr package.

#### Examples

```
>>> s.str.split('_')
>>> s.str.replace('_', '')
```

### pandas.Series.sub

`Series.sub(other, level=None, fill_value=None, axis=0)`

Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

See also:

[`Series.rsub`](#)

## Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```

## pandas.Series.subtract

`Series.subtract` (*other*, *level=None*, *fill\_value=None*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *sub*).

Equivalent to `series - other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

### Returns

**result** [Series]

### See also:

[`Series.rsub`](#)

## Examples

```

>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d     NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b     NaN
d    1.0
e     NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e     NaN
dtype: float64

```

### pandas.Series.sum

`Series.sum(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the sum of the values for the requested axis

#### Parameters

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

#### Returns

**sum** [scalar or Series (if level specified)]

## Examples

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([]).sum() # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

## pandas.Series.swapaxes

`Series.swapaxes` (*axis1*, *axis2*, *copy=True*)  
Interchange axes and swap values axes appropriately

### Returns

**y** [same as input]

## pandas.Series.swaplevel

`Series.swaplevel` (*i=-2*, *j=-1*, *copy=True*)  
Swap levels *i* and *j* in a MultiIndex

**Parameters** *i*, *j*: int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

### Returns

**swapped** [Series]

**.. versionchanged:: 0.18.1**

The indexes *i* and *j* are now optional, and default to the two innermost levels of the index.

## pandas.Series.tail

`Series.tail` (*n=5*)  
Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

**Parameters** `n` : int, default 5

Number of rows to select.

**Returns** type of caller

The last  $n$  rows of the caller object.

**See also:**

`pandas.DataFrame.head` The first  $n$  rows of the caller object.

## Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0 alligator
1      bee
2   falcon
3     lion
4   monkey
5   parrot
6    shark
7    whale
8    zebra
```

Viewing the last 5 lines

```
>>> df.tail()
   animal
4 monkey
5 parrot
6  shark
7  whale
8  zebra
```

Viewing the last  $n$  lines (three in this case)

```
>>> df.tail(3)
   animal
6  shark
7  whale
8  zebra
```

## pandas.Series.take

`Series.take` (*indices*, *axis=0*, *convert=None*, *is\_copy=True*, *\*\*kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters** *indices* : array-like

An array of ints indicating which positions to take.

**axis** : {0 or 'index', 1 or 'columns', None}, default 0

The axis on which to select elements. 0 means that we are selecting rows, 1 means that we are selecting columns.

**convert** : bool, default True

Whether to convert negative indices into positive ones. For example, -1 would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

**is\_copy** : bool, default True

Whether to return a copy of the original object or not.

**\*\*kwargs**

For compatibility with `numpy.take()`. Has no effect on the output.

**Returns taken** : type of caller

An array-like containing the elements taken from the object.

**See also:**

[`DataFrame.loc`](#) Select a subset of a DataFrame by labels.

[`DataFrame.iloc`](#) Select a subset of a DataFrame by positions.

[`numpy.take`](#) Take elements from an array along an axis.

## Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
   name  class  max_speed
0  falcon   bird    389.0
2  parrot   bird     24.0
3   lion  mammal     80.5
1  monkey  mammal      NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
   name  class  max_speed
0  falcon   bird    389.0
1  monkey  mammal      NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
      class  max_speed
0    bird      389.0
2    bird       24.0
3  mammal       80.5
1  mammal        NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
      name  class  max_speed
1  monkey  mammal        NaN
3    lion  mammal       80.5
```

### pandas.Series.to\_clipboard

`Series.to_clipboard(excel=True, sep=None, **kwargs)`

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

**Parameters** `excel` : bool, default True

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

`sep` : str, default '\t'

Field delimiter.

**\*\*kwargs**

These parameters will be passed to `DataFrame.to_csv`.

**See also:**

[`DataFrame.to\_csv`](#) Write a `DataFrame` to a comma-separated values (csv) file.

[`read\_clipboard`](#) Read text from clipboard and pass to `read_table`.

### Notes

Requirements for your platform.

- Linux : `xclip`, or `xsel` (with `gtk` or `PyQt4` modules)
- Windows : none
- OS X : none

### Examples

Copy the contents of a `DataFrame` to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword *index* and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

### pandas.Series.to\_csv

`Series.to_csv` (*path=None*, *index=True*, *sep=' '*, *na\_rep=""*, *float\_format=None*, *header=False*, *index\_label=None*, *mode='w'*, *encoding=None*, *compression=None*, *date\_format=None*, *decimal='.'*)

Write Series to a comma-separated values (csv) file

**Parameters** *path* : string or file handle, default None

File path or object, if None is provided the result is returned as a string.

**na\_rep** : string, default “

Missing data representation

**float\_format** : string, default None

Format string for floating point numbers

**header** : boolean, default False

Write out series name

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**mode** [Python write mode, default ‘w’]

**sep** : character, default “,”

Field delimiter for the output file.

**encoding** : string, optional

a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3

**compression** : string, optional



A string representing the compression to use in the output file. Allowed values are 'gzip', 'bz2', 'zip', 'xz'. This input is only used when the first argument is a filename.

**date\_format:** string, default None

Format string for datetime objects.

**decimal:** string, default '.'

Character recognized as decimal separator. E.g. use ',' for European data

### pandas.Series.to\_dense

`Series.to_dense()`

Return dense representation of NDFrame (as opposed to sparse)

### pandas.Series.to\_dict

`Series.to_dict(into=<class 'dict'>)`

Convert Series to {label -> value} dict or dict-like object.

**Parameters** `into` : class, default dict

The collections.Mapping subclass to use as the return object. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

**Returns**

**value\_dict** [collections.Mapping]

### Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_dict()
{0: 1, 1: 2, 2: 3, 3: 4}
>>> from collections import OrderedDict, defaultdict
>>> s.to_dict(OrderedDict)
OrderedDict([(0, 1), (1, 2), (2, 3), (3, 4)])
>>> dd = defaultdict(list)
>>> s.to_dict(dd)
defaultdict(<type 'list'>, {0: 1, 1: 2, 2: 3, 3: 4})
```

### pandas.Series.to\_excel

`Series.to_excel(excel_writer, sheet_name='Sheet1', na_rep="", float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True)`

Write Series to an excel sheet

New in version 0.20.0.

**Parameters** **excel\_writer** : string or ExcelWriter object

File path or existing ExcelWriter

**sheet\_name** : string, default 'Sheet1'

Name of sheet which will contain DataFrame

**na\_rep** : string, default ''

Missing data representation

**float\_format** : string, default None

Format string for floating point numbers

**columns** : sequence, optional

Columns to write

**header** : boolean or list of string, default True

Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**startrow** :

upper left cell row to dump data frame

**startcol** :

upper left cell column to dump data frame

**engine** : string, default None

write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**merge\_cells** : boolean, default True

Write MultiIndex and Hierarchical Rows as merged cells.

**encoding**: string, default None

encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

**inf\_rep** : string, default 'inf'

Representation for infinity (there is no native representation for infinity in Excel)

**freeze\_panes** : tuple of integer (length 2), default None

Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

## Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer, 'Sheet1')
>>> df2.to_excel(writer, 'Sheet2')
>>> writer.save()
```

For compatibility with `to_csv`, `to_excel` serializes lists and dicts to strings before writing.

## pandas.Series.to\_frame

`Series.to_frame` (*name=None*)

Convert Series to DataFrame

**Parameters** `name` : object, default None

The passed name should substitute for the series name (if it has one).

**Returns**

`data_frame` [DataFrame]

## pandas.Series.to\_hdf

`Series.to_hdf` (*path\_or\_buf, key, \*\*kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the [user guide](#).

**Parameters** `path_or_buf` : str or pandas.HDFStore

File path or HDFStore object.

**key** : str

Identifier for the group in the store.

**mode** : {'a', 'w', 'r+'}, default 'a'

Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

**format** : {'fixed', 'table'}, default 'fixed'

Possible values:

- ‘fixed’: Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- ‘table’: Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

**append** : bool, default False

For Table formats, append the input data to the existing.

**data\_columns** : list of columns or True, optional

List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See [Query via Data Columns](#). Applicable only to format=‘table’.

**complevel** : {0-9}, optional

Specifies a compression level for data. A value of 0 disables compression.

**complib** : {‘zlib’, ‘lzo’, ‘bzip2’, ‘blosc’}, default ‘zlib’

Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: ‘blosc:blosclz’): {‘blosc:blosclz’, ‘blosc:lz4’, ‘blosc:lz4hc’, ‘blosc:snappy’, ‘blosc:zlib’, ‘blosc:zstd’}. Specifying a compression library which is not available issues a ValueError.

**fletcher32** : bool, default False

If applying compression use the fletcher32 checksum.

**dropna** : bool, default False

If true, ALL nan rows will not be written to store.

**errors** : str, default ‘strict’

Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

See also:

**DataFrame.read\_hdf** Read from HDF file.

**DataFrame.to\_parquet** Write a DataFrame to the binary parquet format.

**DataFrame.to\_sql** Write to a sql table.

**DataFrame.to\_feather** Write out feather-format for DataFrames.

**DataFrame.to\_csv** Write out to a csv file.

## Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                   index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

### pandas.Series.to\_json

`Series.to_json(path_or_buf=None, orient=None, date_format=None, double_precision=10, force_ascii=True, date_unit='ms', default_handler=None, lines=False, compression=None, index=True)`  
Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

**Parameters** `path_or_buf` : string or file handle, optional

File path or object. If not specified, the result is returned as a string.

**orient** : string

Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split', 'records', 'index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split', 'records', 'index', 'columns', 'values'}
- The format of the JSON string
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, ... , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}

- ‘values’ : just the values array
- ‘table’ : dict like {‘schema’: {schema}, ‘data’: {data}} describing the data, and the data component is like `orient=‘records’`.

Changed in version 0.20.0.

**date\_format** : {None, ‘epoch’, ‘iso’}

Type of date conversion. ‘epoch’ = epoch milliseconds, ‘iso’ = ISO8601. The default depends on the *orient*. For `orient=‘table’`, the default is ‘iso’. For all other orients, the default is ‘epoch’.

**double\_precision** : int, default 10

The number of decimal places to use when encoding floating point values.

**force\_ascii** : boolean, default True

Force encoded string to be ASCII.

**date\_unit** : string, default ‘ms’ (milliseconds)

The time unit to encode to, governs timestamp and ISO8601 precision. One of ‘s’, ‘ms’, ‘us’, ‘ns’ for second, millisecond, microsecond, and nanosecond respectively.

**default\_handler** : callable, default None

Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

**lines** : boolean, default False

If ‘orient’ is ‘records’ write out line delimited json format. Will throw ValueError if incorrect ‘orient’ since others are not list like.

New in version 0.19.0.

**compression** : {None, ‘gzip’, ‘bz2’, ‘zip’, ‘xz’}

A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

**index** : boolean, default True

Whether to include the index values in the JSON string. Not including the index (`index=False`) is only supported when orient is ‘split’ or ‘table’.

New in version 0.23.0.

#### See also:

[`pandas.read\_json`](#)

#### Examples

```
>>> df = pd.DataFrame([[‘a’, ‘b’], [‘c’, ‘d’]],
...                    index=[‘row 1’, ‘row 2’],
...                    columns=[‘col 1’, ‘col 2’])
>>> df.to_json(orient=‘split’)
```

(continues on next page)

(continued from previous page)

```
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]]'
```

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{ "col 1": "a", "col 2": "b"}, { "col 1": "c", "col 2": "d"}]'
```

Encoding/decoding a Dataframe using 'index' formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using 'columns' formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using 'values' formatted JSON:

```
>>> df.to_json(orient='values')
'[[["a","b"],["c","d"]]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
  "primaryKey": "index",
  "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
            {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

## pandas.Series.to\_latex

`Series.to_latex` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, bold\_rows=False, column\_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires `\usepackage{booktabs}`.

Changed in version 0.20.2: Added to Series

*to\_latex*-specific options:

**bold\_rows** [boolean, default False] Make the row labels bold in the output

**column\_format** [str, default None] The columns format as specified in [LaTeX table format](#) e.g 'rcl' for 3 columns

**longtable** [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a `\usepackage{longtable}` to your LaTeX preamble.

**escape** [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

**encoding** [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

**decimal** [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

**multicolumn** [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

**multicolumn\_format** [str, default 'l'] The alignment for multicolumns, similar to *column\_format*. The default will be read from the config module.

New in version 0.20.0.

**multirow** [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a `\usepackage{multirow}` to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

### **pandas.Series.to\_msgpack**

`Series.to_msgpack` (*path\_or\_buf=None, encoding='utf-8', \*\*kwargs*)  
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

**Parameters** **path** : string File path, buffer-like, or None

if None, return generated string

**append** : boolean whether to append to an existing msgpack

(default is False)

**compress** : type of compressor (zlib or blosc), default to None (no compression)

### **pandas.Series.to\_period**

`Series.to_period` (*freq=None, copy=True*)

Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

**Parameters**

**freq** [string, default]

**Returns**

**ts** [Series with PeriodIndex]



**pandas.Series.to\_pickle**`Series.to_pickle` (*path*, *compression*='infer', *protocol*=4)

Pickle (serialize) object to file.

**Parameters** *path* : str

File path where the pickled object will be stored.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

**protocol** : int

Int which indicates which protocol should be used by the pickler, default HIGHEST\_PROTOCOL (see [R27] paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python &gt;= 3.0, 3 is a valid value. For Python &gt;= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST\_PROTOCOL.

New in version 0.21.0.

**See also:**`read_pickle` Load pickled pandas object (or any object) from file.`DataFrame.to_hdf` Write DataFrame to an HDF5 file.`DataFrame.to_sql` Write DataFrame to a SQL database.`DataFrame.to_parquet` Write a DataFrame to the binary parquet format.**Examples**

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

## pandas.Series.to\_sparse

`Series.to_sparse` (*kind='block', fill\_value=None*)  
Convert Series to SparseSeries

### Parameters

**kind** [{ 'block', 'integer' }]

**fill\_value** [float, defaults to NaN (missing)]

### Returns

**sp** [SparseSeries]

## pandas.Series.to\_sql

`Series.to_sql` (*name, con, schema=None, if\_exists='fail', index=True, index\_label=None, chunk-size=None, dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [\[R28\]](#) are supported. Tables can be newly created, appended to, or overwritten.

**Parameters** **name** : string

Name of SQL table.

**con** : sqlalchemy.engine.Engine or sqlite3.Connection

Using SQLAlchemy makes it possible to use any DB supported by that library.  
Legacy support is provided for sqlite3.Connection objects.

**schema** : string, optional

Specify the schema (if database flavor supports this). If None, use default schema.

**if\_exists** : { 'fail', 'replace', 'append' }, default 'fail'

How to behave if the table already exists.

- fail: Raise a ValueError.
- replace: Drop the table before inserting new values.
- append: Insert new values to the existing table.

**index** : boolean, default True

Write DataFrame index as a column. Uses *index\_label* as the column name in the table.

**index\_label** : string or sequence, default None

Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**chunksize** : int, optional

Rows will be written in batches of this size at a time. By default, all rows will be written at once.

**dtype** : dict, optional

Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

#### Raises **ValueError**

When the table already exists and *if\_exists* is 'fail' (the default).

See also:

`pandas.read_sql` read a DataFrame from a table

## References

[R28], [R29]

## Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
   name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just df1.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...           index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
   A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...          dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

### **pandas.Series.to\_string**

`Series.to_string(buf=None, na_rep='NaN', float_format=None, header=True, index=True, length=False, dtype=False, name=False, max_rows=None)`

Render a string representation of the Series

**Parameters** `buf` : StringIO-like, optional

buffer to write to

**na\_rep** : string, optional

string representation of NaN to use, default 'NaN'

**float\_format** : one-parameter function, optional

formatter function to apply to columns' elements if they are floats default None

**header: boolean, default True**

Add the Series header (index name)

**index** : bool, optional

Add index (row) labels, default True

**length** : boolean, default False

Add the Series length

**dtype** : boolean, default False

Add the Series dtype

**name** : boolean, default False

Add the Series name if not None

**max\_rows** : int, optional

Maximum number of rows to show before truncating. If None, show all.

**Returns**

**formatted** [string (if not buffer passed)]

**pandas.Series.to\_timestamp**`Series.to_timestamp(freq=None, how='start', copy=True)`Cast to DatetimeIndex of timestamps, at *beginning* of period**Parameters** `freq` : string, default frequency of PeriodIndex

Desired frequency

**how** : {'s', 'e', 'start', 'end'}

Convention for converting period to timestamp; start of period vs. end

**Returns**`ts` [Series with DatetimeIndex]**pandas.Series.to\_xarray**`Series.to_xarray()`

Return an xarray object from the pandas object.

**Returns****a DataArray for a Series****a Dataset for a DataFrame****a DataArray for higher dims****Notes**See the [xarray docs](#)**Examples**

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                        'B' : ['foo', 'bar', 'foo'],
                        'C' : np.arange(4., 7)})

>>> df
   A  B  C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index      (index) int64 0 1 2
Data variables:
  A            (index) int64 1 1 2
  B            (index) object 'foo' 'bar' 'foo'
  C            (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                        'B' : ['foo', 'bar', 'foo'],
                        'C' : np.arange(4.,7)}
                        ).set_index(['B','A'])

>>> df
```

	A	C
foo 1	1	4.0
bar 1	1	5.0
foo 2	2	6.0

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B        (B) object 'bar' 'foo'
  * A        (A) int64 1 2
Data variables:
  C          (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])

>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
         [ 2,  3],
         [ 4,  5]],
       [[ 6,  7],
         [ 8,  9],
         [10, 11]],
       [[12, 13],
         [14, 15],
         [16, 17]],
       [[18, 19],
         [20, 21],
         [22, 23]]])
Coordinates:
  * items      (items) object 'A' 'B' 'C' 'D'
  * major_axis (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
  # noqa
  * minor_axis (minor_axis) object 'first' 'second'
```

## pandas.Series.tolist

`Series.tolist()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

**See also:**

`numpy.ndarray.tolist`

## pandas.Series.transform

`Series.transform(func, *args, **kwargs)`

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

**Parameters** `func` : callable, string, dictionary, or list of string/callables

To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns**

**transformed** [NDFrame]

**See also:**

`pandas.NDFrame.aggregate`, `pandas.NDFrame.apply`

## Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                    index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan
```

```
>>> df.transform(lambda x: (x - x.mean()) / x.std())
```

	A	B	C
2000-01-01	0.579457	1.236184	0.123424
2000-01-02	0.370357	-0.605875	-1.231325
2000-01-03	1.455756	-0.277446	0.288967
2000-01-04	NaN	NaN	NaN
2000-01-05	NaN	NaN	NaN
2000-01-06	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN
2000-01-08	-0.498658	1.274522	1.642524
2000-01-09	-0.540524	-1.012676	-0.828968
2000-01-10	-1.366388	-0.614710	0.005378

### pandas.Series.transpose

`Series.transpose(*args, **kwargs)`  
return the transpose, which is by definition self

### pandas.Series.truediv

`Series.truediv(other, level=None, fill_value=None, axis=0)`  
Floating division of series and other, element-wise (binary operator *truediv*).

Equivalent to `series / other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series or scalar value]

**fill\_value** : None or float value, default None (NaN)

Fill existing missing (NaN) values, and any new element needed for successful Series alignment, with this value before computation. If data in both corresponding Series locations is missing the result will be missing

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

#### Returns

**result** [Series]

#### See also:

[`Series.rtruediv`](#)

### Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b, fill_value=0)
a    2.0
b    1.0
c    1.0
d    1.0
e    NaN
dtype: float64
```



**pandas.Series.truncate**

`Series.truncate` (*before=None, after=None, axis=None, copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

**Parameters** **before** : date, string, int

Truncate all rows before this index value.

**after** : date, string, int

Truncate all rows after this index value.

**axis** : {0 or 'index', 1 or 'columns'}, optional

Axis to truncate. Truncates the index (rows) by default.

**copy** : boolean, default is True,

Return a copy of the truncated section.

**Returns** **type of caller**

The truncated Series or DataFrame.

**See also:**

[`DataFrame.loc`](#) Select a subset of a DataFrame by label.

[`DataFrame.iloc`](#) Select a subset of a DataFrame by position.

**Notes**

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

**Examples**

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a `DatetimeIndex` containing only dates, we can specify *before* and *after* as strings. They will be coerced to `Timestamps` before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
                A
2016-01-10 23:59:55  1
2016-01-10 23:59:56  1
2016-01-10 23:59:57  1
```

(continues on next page)

(continued from previous page)

```
2016-01-10 23:59:58 1
2016-01-10 23:59:59 1
```

**pandas.Series.tshift****Series.tshift** (*periods=1, freq=None, axis=0*)

Shift the time index, using the index's frequency if available.

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, default None

Increment to use from the tseries module or time rule (e.g. 'EOM')

**axis** : int or basestring

Corresponds to the axis that contains the Index

**Returns****shifted** [NDFrame]**Notes**

If freq is not specified then tries to use the freq or inferred\_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

**pandas.Series.tz\_convert****Series.tz\_convert** (*tz, axis=0, level=None, copy=True*)

Convert tz-aware axis to target time zone.

**Parameters****tz** [string or pytz.timezone object]**axis** [the axis to convert]**level** : int, str, default None

If axis is a MultiIndex, convert a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**Raises** **TypeError**

If the axis is tz-naive.

**pandas.Series.tz\_localize****Series.tz\_localize** (*tz, axis=0, level=None, copy=True, ambiguous='raise'*)

Localize tz-naive TimeSeries to target time zone.

**Parameters**

**tz** [string or pytz.timezone object]

**axis** [the axis to localize]

**level** : int, str, default None

If axis is a MultiIndex, localize a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**ambiguous** : 'infer', bool-ndarray, 'NaT', default 'raise'

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

**Raises TypeError**

If the TimeSeries is tz-aware and tz is not None.

**pandas.Series.unique**

`Series.unique()`

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

**Returns ndarray or Categorical**

The unique values returned as a NumPy array. In case of categorical data type, returned as a Categorical.

**See also:**

[`pandas.unique`](#) top-level unique method for any 1-d array-like object.

[`Index.unique`](#) return Index with unique values from an Index object.

**Examples**

```
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...           for _ in range(3)]).unique()
array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
      dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
...                           ordered=True)).unique()
[b, a, c]
Categories (3, object): [a < b < c]
```

## pandas.Series.unstack

`Series.unstack (level=-1, fill_value=None)`

Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame. The level involved will automatically get sorted.

**Parameters** `level` : int, string, or list of these, default last level

Level(s) to unstack, can pass level name

`fill_value` : replace NaN with this value if the unstack produces missing values

New in version 0.18.0.

**Returns**

`unstacked` [DataFrame]

## Examples

```
>>> s = pd.Series([1, 2, 3, 4],
...               index=pd.MultiIndex.from_product([['one', 'two'], ['a', 'b']]))
>>> s
one  a    1
     b    2
two  a    3
     b    4
dtype: int64
```

```
>>> s.unstack(level=-1)
     a  b
one  1  2
two  3  4
```

```
>>> s.unstack(level=0)
     one  two
a      1    3
b      2    4
```

## pandas.Series.update

`Series.update(other)`

Modify Series in place using non-NA values from passed Series. Aligns on index

### Parameters

**other** [Series]

### Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6]))
>>> s
0    4
1    5
2    6
dtype: int64
```

```
>>> s = pd.Series(['a', 'b', 'c'])
>>> s.update(pd.Series(['d', 'e'], index=[0, 2]))
>>> s
0    d
1    b
2    e
dtype: object
```

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, 5, 6, 7, 8]))
>>> s
0    4
1    5
2    6
dtype: int64
```

If other contains NaNs the corresponding values are not updated in the original Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.update(pd.Series([4, np.nan, 6]))
>>> s
0    4
1    2
2    6
dtype: int64
```

## pandas.Series.valid

`Series.valid(inplace=False, **kwargs)`

Return Series without null values.

Deprecated since version 0.23.0: Use `Series.dropna()` instead.

**pandas.Series.value\_counts**

`Series.value_counts` (*normalize=False, sort=True, ascending=False, bins=None, dropna=True*)

Returns object containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

**Parameters** **normalize** : boolean, default False

If True then the object returned will contain the relative frequencies of the unique values.

**sort** : boolean, default True

Sort by values

**ascending** : boolean, default False

Sort in ascending order

**bins** : integer, optional

Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data

**dropna** : boolean, default True

Don't include counts of NaN.

**Returns**

**counts** [Series]

**pandas.Series.var**

`Series.var` (*axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the `ddof` argument

**Parameters**

**axis** [{index (0)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a scalar

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**var** [scalar or Series (if level specified)]

**pandas.Series.view**

`Series.view` (*dtype=None*)

Create a new view of the Series.

This function will return a new Series with a view of the same underlying values in memory, optionally reinterpreted with a new data type. The new data type must preserve the same size in bytes as to not cause index misalignment.

**Parameters** **dtype** : data type

Data type object or one of their string representations.

**Returns** **Series**

A new Series object as a view of the same data in memory.

**See also:**

**numpy.ndarray.view** Equivalent numpy function to create a new view of the same data in memory.

**Notes**

Series are instantiated with `dtype=float64` by default. While `numpy.ndarray.view()` will return a view with the same data type as the original array, `Series.view()` (without specified `dtype`) will try using `float64` and may fail if the original data type size in bytes is not the same.

**Examples**

```
>>> s = pd.Series([-2, -1, 0, 1, 2], dtype='int8')
>>> s
0    -2
1    -1
2     0
3     1
4     2
dtype: int8
```

The 8 bit signed integer representation of `-1` is `0b11111111`, but the same bytes represent 255 if read as an 8 bit unsigned integer:

```
>>> us = s.view('uint8')
>>> us
0    254
1    255
2     0
3     1
4     2
dtype: uint8
```

The views share the same underlying values:



```

>>> us[0] = 128
>>> s
0    -128
1      -1
2       0
3       1
4       2
dtype: int8

```

### pandas.Series.where

`Series.where(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)`

Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

**Parameters** *cond* : boolean NDFrame, array-like, or callable

Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as *cond*.

**other** : scalar, NDFrame, or callable

Entries where *cond* is False are replaced with corresponding value from *other*. If *other* is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as *other*.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

**axis** [alignment axis if needed, default None]

**level** [alignment level if needed, default None]

**errors** : str, {'raise', 'ignore'}, default 'raise'

- *raise* : allow exceptions to be raised
- *ignore* : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

### Returns

**wh** [same type as caller]

**See also:**

`DataFrame.mask()`

## Notes

The `where` method is an application of the if-then idiom. For each element in the calling `DataFrame`, if `cond` is `True` the element is used; otherwise the corresponding element from the `DataFrame` `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in [indexing](#).

## Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1     NaN
2     NaN
3     NaN
4     NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A      B
0  True  True
1  True  True
2  True  True
```

(continues on next page)

(continued from previous page)

```

3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True

```

**pandas.Series.xs****Series.xs** (*key*, *axis=0*, *level=None*, *drop\_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (*axis=0*).

**Parameters** **key** : object

Some label contained in the index, or partially in a MultiIndex

**axis** : int, default 0

Axis to retrieve cross-section on

**level** : object, defaults to first *n* levels (*n=1* or *len(key)*)

In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

**drop\_level** : boolean, default True

If False, returns object with same levels as self.

**Returns****xs** [Series or DataFrame]**Notes**

*xs* is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see [MultiIndex Slicers](#)

**Examples**

```

>>> df
      A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
A    4
B    5
C    2

```

(continues on next page)

(continued from previous page)

```
Name: a
>>> df.xs('C', axis=1)
a      2
b      9
c      3
Name: C
```

```
>>> df
      first second third      A  B  C  D
bar    one     1      4  1  8  9
      two     1      7  5  5  0
baz    one     1      6  6  8  0
      three    2      5  3  5  3
>>> df.xs(('baz', 'three'))
      A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
      A  B  C  D
first third
bar    1      4  1  8  9
baz    1      6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
      A  B  C  D
second
three    5  3  5  3
```

### 34.3.2 Attributes

#### Axes

<code>Series.index</code>	The index (axis labels) of the Series.
<code>Series.values</code>	Return Series as ndarray or ndarray-like depending on the dtype
<code>Series.dtype</code>	return the dtype object of the underlying data
<code>Series.ftype</code>	return if the data is sparsedense
<code>Series.shape</code>	return a tuple of the shape of the underlying data
<code>Series.nbytes</code>	return the number of bytes in the underlying data
<code>Series.ndim</code>	return the number of dimensions of the underlying data, by definition 1
<code>Series.size</code>	return the number of elements in the underlying data
<code>Series.strides</code>	return the strides of the underlying data
<code>Series.itemsize</code>	return the size of the dtype of the item of the underlying data
<code>Series.base</code>	return the base object if the memory of the underlying data is shared
<code>Series.T</code>	return the transpose, which is by definition self
<code>Series.memory_usage([index, deep])</code>	Return the memory usage of the Series.
<code>Series.hasnans</code>	return if I have any nans; enables various perf speedups

Continued on next page

Table 27 – continued from previous page

<code>Series.flags</code>	
<code>Series.empty</code>	
<code>Series.dtypes</code>	return the dtype object of the underlying data
<code>Series.ftypes</code>	return if the data is sparsedense
<code>Series.data</code>	return the data pointer of the underlying data
<code>Series.is_copy</code>	
<code>Series.name</code>	
<code>Series.put(*args, **kwargs)</code>	Applies the <i>put</i> method to its <i>values</i> attribute if it has one.

### 34.3.2.1 pandas.Series.empty

`Series.empty`

### 34.3.2.2 pandas.Series.is\_copy

`Series.is_copy`

### 34.3.2.3 pandas.Series.name

`Series.name`

## 34.3.3 Conversion

<code>Series.astype(dtype[, copy, errors])</code>	Cast a pandas object to a specified dtype <i>dtype</i> .
<code>Series.infer_objects()</code>	Attempt to infer better dtypes for object columns.
<code>Series.convert_objects([convert_dates, ...])</code>	(DEPRECATED) Attempt to infer better dtype for object columns.
<code>Series.copy([deep])</code>	Make a copy of this object's indices and data.
<code>Series.bool()</code>	Return the bool of a single element PandasObject.
<code>Series.to_period([freq, copy])</code>	Convert Series from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)
<code>Series.to_timestamp([freq, how, copy])</code>	Cast to datetimelindex of timestamps, at <i>beginning</i> of period
<code>Series.tolist()</code>	Return a list of the values.
<code>Series.get_values()</code>	same as values (but handles sparseness conversions); is a view

## 34.3.4 Indexing, iteration

<code>Series.get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>Series.at</code>	Access a single value for a row/column label pair.
<code>Series.iat</code>	Access a single value for a row/column pair by integer position.

Continued on next page

Table 29 – continued from previous page

<code>Series.loc</code>	Access a group of rows and columns by label(s) or a boolean array.
<code>Series.iloc</code>	Purely integer-location based indexing for selection by position.
<code>Series.__iter__()</code>	Return an iterator of the values.
<code>Series.iteritems()</code>	Lazily iterate over (index, value) tuples
<code>Series.items()</code>	Lazily iterate over (index, value) tuples
<code>Series.keys()</code>	Alias for index
<code>Series.pop(item)</code>	Return item and drop from frame.
<code>Series.item()</code>	return the first element of the underlying data as a python scalar
<code>Series.xs(key[, axis, level, drop_level])</code>	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

#### 34.3.4.1 pandas.Series.\_\_iter\_\_

`Series.__iter__()`

Return an iterator of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the [indexing documentation](#).

### 34.3.5 Binary operator functions

<code>Series.add(other[, level, fill_value, axis])</code>	Addition of series and other, element-wise (binary operator <i>add</i> ).
<code>Series.sub(other[, level, fill_value, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>Series.mul(other[, level, fill_value, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>Series.div(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>Series.truediv(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>Series.floordiv(other[, level, fill_value, axis])</code>	Integer division of series and other, element-wise (binary operator <i>floordiv</i> ).
<code>Series.mod(other[, level, fill_value, axis])</code>	Modulo of series and other, element-wise (binary operator <i>mod</i> ).
<code>Series.pow(other[, level, fill_value, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>pow</i> ).
<code>Series.radd(other[, level, fill_value, axis])</code>	Addition of series and other, element-wise (binary operator <i>radd</i> ).
<code>Series.rsub(other[, level, fill_value, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>rsub</i> ).
<code>Series.rmul(other[, level, fill_value, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>rmul</i> ).
<code>Series.rdiv(other[, level, fill_value, axis])</code>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).

Continued on next page

Table 30 – continued from previous page

<code>Series.rtruediv</code> (other[, level, fill_value, axis])	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<code>Series.rfloordiv</code> (other[, level, fill_value, ...])	Integer division of series and other, element-wise (binary operator <i>rfloordiv</i> ).
<code>Series.rmod</code> (other[, level, fill_value, axis])	Modulo of series and other, element-wise (binary operator <i>rmod</i> ).
<code>Series.rpow</code> (other[, level, fill_value, axis])	Exponential power of series and other, element-wise (binary operator <i>rpow</i> ).
<code>Series.combine</code> (other, func[, fill_value])	Perform elementwise binary operation on two Series using given function with optional fill value when an index is missing from one Series or the other
<code>Series.combine_first</code> (other)	Combine Series values, choosing the calling Series's values first.
<code>Series.round</code> ([decimals])	Round each value in a Series to the given number of decimals.
<code>Series.lt</code> (other[, level, fill_value, axis])	Less than of series and other, element-wise (binary operator <i>lt</i> ).
<code>Series.gt</code> (other[, level, fill_value, axis])	Greater than of series and other, element-wise (binary operator <i>gt</i> ).
<code>Series.le</code> (other[, level, fill_value, axis])	Less than or equal to of series and other, element-wise (binary operator <i>le</i> ).
<code>Series.ge</code> (other[, level, fill_value, axis])	Greater than or equal to of series and other, element-wise (binary operator <i>ge</i> ).
<code>Series.ne</code> (other[, level, fill_value, axis])	Not equal to of series and other, element-wise (binary operator <i>ne</i> ).
<code>Series.eq</code> (other[, level, fill_value, axis])	Equal to of series and other, element-wise (binary operator <i>eq</i> ).
<code>Series.product</code> ([axis, skipna, level, ...])	Return the product of the values for the requested axis
<code>Series.dot</code> (other)	Matrix multiplication with DataFrame or inner-product with Series objects.

### 34.3.6 Function application, GroupBy & Window

<code>Series.apply</code> (func[, convert_dtype, args])	Invoke function on values of Series.
<code>Series.agg</code> (func[, axis])	Aggregate using one or more operations over the specified axis.
<code>Series.aggregate</code> (func[, axis])	Aggregate using one or more operations over the specified axis.
<code>Series.transform</code> (func, *args, **kwargs)	Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
<code>Series.map</code> (arg[, na_action])	Map values of Series using input correspondence (a dict, Series, or function).
<code>Series.groupby</code> ([by, axis, level, as_index, ...])	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
<code>Series.rolling</code> (window[, min_periods, ...])	Provides rolling window calculations.
<code>Series.expanding</code> ([min_periods, center, axis])	Provides expanding transformations.
<code>Series.ewm</code> ([com, span, halflife, alpha, ...])	Provides exponential weighted functions
<code>Series.pipe</code> (func, *args, **kwargs)	Apply func(self, *args, **kwargs)

### 34.3.7 Computations / Descriptive Stats

<code>Series.abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>Series.all([axis, bool_only, skipna, level])</code>	Return whether all elements are True, potentially over an axis.
<code>Series.any([axis, bool_only, skipna, level])</code>	Return whether any element is True over requested axis.
<code>Series.autocorr([lag])</code>	Lag-N autocorrelation
<code>Series.between(left, right[, inclusive])</code>	Return boolean Series equivalent to <code>left &lt;= series &lt;= right</code> .
<code>Series.clip([lower, upper, axis, inplace])</code>	Trim values at input threshold(s).
<code>Series.clip_lower(threshold[, axis, inplace])</code>	Return copy of the input with values below a threshold truncated.
<code>Series.clip_upper(threshold[, axis, inplace])</code>	Return copy of input with values above given value(s) truncated.
<code>Series.corr(other[, method, min_periods])</code>	Compute correlation with <i>other</i> Series, excluding missing values
<code>Series.count([level])</code>	Return number of non-NA/null observations in the Series
<code>Series.cov(other[, min_periods])</code>	Compute covariance with Series, excluding missing values
<code>Series.cummax([axis, skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>Series.cummin([axis, skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>Series.cumprod([axis, skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>Series.cumsum([axis, skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>Series.describe([percentiles, include, exclude])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>Series.diff([periods])</code>	First discrete difference of element.
<code>Series.factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>Series.kurt([axis, skipna, level, numeric_only])</code>	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>Series.mad([axis, skipna, level])</code>	Return the mean absolute deviation of the values for the requested axis
<code>Series.max([axis, skipna, level, numeric_only])</code>	This method returns the maximum of the values in the object.
<code>Series.mean([axis, skipna, level, numeric_only])</code>	Return the mean of the values for the requested axis
<code>Series.median([axis, skipna, level, ...])</code>	Return the median of the values for the requested axis
<code>Series.min([axis, skipna, level, numeric_only])</code>	This method returns the minimum of the values in the object.
<code>Series.mode()</code>	Return the mode(s) of the dataset.
<code>Series.nlargest([n, keep])</code>	Return the largest <i>n</i> elements.
<code>Series.nsmallest([n, keep])</code>	Return the smallest <i>n</i> elements.
<code>Series.pct_change([periods, fill_method, ...])</code>	Percentage change between the current and a prior element.
<code>Series.prod([axis, skipna, level, ...])</code>	Return the product of the values for the requested axis

Continued on next page



Table 32 – continued from previous page

<code>Series.quantile([q, interpolation])</code>	Return value at the given quantile, a la <code>numpy.percentile</code> .
<code>Series.rank([axis, method, numeric_only, ...])</code>	Compute numerical data ranks (1 through n) along axis.
<code>Series.sem([axis, skipna, level, ddof, ...])</code>	Return unbiased standard error of the mean over requested axis.
<code>Series.skew([axis, skipna, level, numeric_only])</code>	Return unbiased skew over requested axis Normalized by N-1
<code>Series.std([axis, skipna, level, ddof, ...])</code>	Return sample standard deviation over requested axis.
<code>Series.sum([axis, skipna, level, ...])</code>	Return the sum of the values for the requested axis
<code>Series.var([axis, skipna, level, ddof, ...])</code>	Return unbiased variance over requested axis.
<code>Series.kurtosis([axis, skipna, level, ...])</code>	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>Series.unique()</code>	Return unique values of Series object.
<code>Series.nunique([dropna])</code>	Return number of unique elements in the object.
<code>Series.is_unique</code>	Return boolean if values in the object are unique
<code>Series.is_monotonic</code>	Return boolean if values in the object are monotonic_increasing
<code>Series.is_monotonic_increasing</code>	Return boolean if values in the object are monotonic_increasing
<code>Series.is_monotonic_decreasing</code>	Return boolean if values in the object are monotonic_decreasing
<code>Series.value_counts([normalize, sort, ...])</code>	Returns object containing counts of unique values.
<code>Series.compound([axis, skipna, level])</code>	Return the compound percentage of the values for the requested axis
<code>Series.nonzero()</code>	Return the <i>integer</i> indices of the elements that are non-zero
<code>Series.ptp([axis, skipna, level, numeric_only])</code>	Returns the difference between the maximum value and the minimum value in the object.

### 34.3.8 Reindexing / Selection / Label manipulation

<code>Series.align(other[, join, axis, level, ...])</code>	Align two objects on their axes with the specified join method for each axis Index
<code>Series.drop([labels, axis, index, columns, ...])</code>	Return Series with specified index labels removed.
<code>Series.drop_duplicates([keep, inplace])</code>	Return Series with duplicate values removed.
<code>Series.duplicated([keep])</code>	Indicate duplicate Series values.
<code>Series.equals(other)</code>	Determines if two NDFrame objects contain the same elements.
<code>Series.first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>Series.head([n])</code>	Return the first <i>n</i> rows.
<code>Series.idxmax([axis, skipna])</code>	Return the row label of the maximum value.
<code>Series.idxmin([axis, skipna])</code>	Return the row label of the minimum value.
<code>Series.isin(values)</code>	Check whether <i>values</i> are contained in Series.
<code>Series.last(offset)</code>	Convenience method for subsetting final periods of time series data based on a date offset.
<code>Series.reindex([index])</code>	Conform Series to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.

Continued on next page

Table 33 – continued from previous page

<code>Series.reindex_like(other[, method, copy, ...])</code>	Return an object with matching indices to myself.
<code>Series.rename([index])</code>	Alter Series index labels or name
<code>Series.rename_axis(mapper[, axis, copy, inplace])</code>	Alter the name of the index or columns.
<code>Series.reset_index([level, drop, name, inplace])</code>	Generate a new DataFrame or Series with the index reset.
<code>Series.sample([n, frac, replace, weights, ...])</code>	Return a random sample of items from an axis of object.
<code>Series.select(crit[, axis])</code>	(DEPRECATED) Return data corresponding to axis labels matching criteria
<code>Series.set_axis(labels[, axis, inplace])</code>	Assign desired index to given axis.
<code>Series.take(indices[, axis, convert, is_copy])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>Series.tail([n])</code>	Return the last <i>n</i> rows.
<code>Series.truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.
<code>Series.where(cond[, other, inplace, axis, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .
<code>Series.mask(cond[, other, inplace, axis, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>Series.add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>Series.add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>Series.filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.

### 34.3.9 Missing data handling

<code>Series.isna()</code>	Detect missing values.
<code>Series.notna()</code>	Detect existing (non-missing) values.
<code>Series.dropna([axis, inplace])</code>	Return a new Series with missing values removed.
<code>Series.fillna([value, method, axis, ...])</code>	Fill NA/NaN values using the specified method
<code>Series.interpolate([method, axis, limit, ...])</code>	Interpolate values according to different methods.

### 34.3.10 Reshaping, sorting

<code>Series.argsort([axis, kind, order])</code>	Overrides ndarray.argsort.
<code>Series.argmin([axis, skipna])</code>	(DEPRECATED) ..
<code>Series.argmax([axis, skipna])</code>	(DEPRECATED) ..
<code>Series.reorder_levels(order)</code>	Rearrange index levels using input order.
<code>Series.sort_values([axis, ascending, ...])</code>	Sort by the values.
<code>Series.sort_index([axis, level, ascending, ...])</code>	Sort Series by index labels.
<code>Series.swaplevel([i, j, copy])</code>	Swap levels <i>i</i> and <i>j</i> in a MultiIndex
<code>Series.unstack([level, fill_value])</code>	Unstack, a.k.a.
<code>Series.searchsorted(value[, side, sorter])</code>	Find indices where elements should be inserted to maintain order.
<code>Series.ravel([order])</code>	Return the flattened underlying data as an ndarray
<code>Series.repeat(repeats, *args, **kwargs)</code>	Repeat elements of an Series.
<code>Series.squeeze([axis])</code>	Squeeze length 1 dimensions.

Continued on next page

Table 35 – continued from previous page

<code>Series.view([dtype])</code>	Create a new view of the Series.
<code>Series.sortlevel([level, ascending, ...])</code>	(DEPRECATED) Sort Series with MultiIndex by chosen level.

### 34.3.11 Combining / joining / merging

<code>Series.append(to_append[, ignore_index, ...])</code>	Concatenate two or more Series.
<code>Series.replace([to_replace, value, inplace, ...])</code>	Replace values given in <i>to_replace</i> with <i>value</i> .
<code>Series.update(other)</code>	Modify Series in place using non-NA values from passed Series.

### 34.3.12 Time series-related

<code>Series.asfreq(freq[, method, how, ...])</code>	Convert TimeSeries to specified frequency.
<code>Series.asof(where[, subset])</code>	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
<code>Series.shift([periods, freq, axis])</code>	Shift index by desired number of periods with an optional time freq
<code>Series.first_valid_index()</code>	Return index for first non-NA/null value.
<code>Series.last_valid_index()</code>	Return index for last non-NA/null value.
<code>Series.resample(rule[, how, axis, ...])</code>	Convenience method for frequency conversion and re-sampling of time series.
<code>Series.tz_convert(tz[, axis, level, copy])</code>	Convert tz-aware axis to target time zone.
<code>Series.tz_localize(tz[, axis, level, copy, ...])</code>	Localize tz-naive TimeSeries to target time zone.
<code>Series.at_time(time[, asof])</code>	Select values at particular time of day (e.g.
<code>Series.between_time(start_time, end_time[, ...])</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>Series.tshift([periods, freq, axis])</code>	Shift the time index, using the index's frequency if available.
<code>Series.slice_shift([periods, axis])</code>	Equivalent to <i>shift</i> without copying data.

### 34.3.13 Datetimelike Properties

`Series.dt` can be used to access the values of the series as datetimelike and return several properties. These can be accessed like `Series.dt.<property>`.

#### Datetime Properties

<code>Series.dt.date</code>	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).
<code>Series.dt.time</code>	Returns numpy array of datetime.time.
<code>Series.dt.year</code>	The year of the datetime
<code>Series.dt.month</code>	The month as January=1, December=12
<code>Series.dt.day</code>	The days of the datetime
<code>Series.dt.hour</code>	The hours of the datetime
<code>Series.dt.minute</code>	The minutes of the datetime
<code>Series.dt.second</code>	The seconds of the datetime

Continued on next page

Table 38 – continued from previous page

<code>Series.dt.microsecond</code>	The microseconds of the datetime
<code>Series.dt.nanosecond</code>	The nanoseconds of the datetime
<code>Series.dt.week</code>	The week ordinal of the year
<code>Series.dt.weekofyear</code>	The week ordinal of the year
<code>Series.dt.dayofweek</code>	The day of the week with Monday=0, Sunday=6
<code>Series.dt.weekday</code>	The day of the week with Monday=0, Sunday=6
<code>Series.dt.dayofyear</code>	The ordinal day of the year
<code>Series.dt.quarter</code>	The quarter of the date
<code>Series.dt.is_month_start</code>	Logical indicating if first day of month (defined by frequency)
<code>Series.dt.is_month_end</code>	Indicator for whether the date is the last day of the month.
<code>Series.dt.is_quarter_start</code>	Indicator for whether the date is the first day of a quarter.
<code>Series.dt.is_quarter_end</code>	Indicator for whether the date is the last day of a quarter.
<code>Series.dt.is_year_start</code>	Indicate whether the date is the first day of a year.
<code>Series.dt.is_year_end</code>	Indicate whether the date is the last day of the year.
<code>Series.dt.is_leap_year</code>	Boolean indicator if the date belongs to a leap year.
<code>Series.dt.daysinmonth</code>	The number of days in the month
<code>Series.dt.days_in_month</code>	The number of days in the month
<code>Series.dt.tz</code>	
<code>Series.dt.freq</code>	

### 34.3.13.1 pandas.Series.dt.date

`Series.dt.date`

Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).

### 34.3.13.2 pandas.Series.dt.time

`Series.dt.time`

Returns numpy array of datetime.time. The time part of the Timestamps.

### 34.3.13.3 pandas.Series.dt.year

`Series.dt.year`

The year of the datetime

### 34.3.13.4 pandas.Series.dt.month

`Series.dt.month`

The month as January=1, December=12

### 34.3.13.5 pandas.Series.dt.day

`Series.dt.day`

The days of the datetime

#### 34.3.13.6 pandas.Series.dt.hour

`Series.dt.hour`

The hours of the datetime

#### 34.3.13.7 pandas.Series.dt.minute

`Series.dt.minute`

The minutes of the datetime

#### 34.3.13.8 pandas.Series.dt.second

`Series.dt.second`

The seconds of the datetime

#### 34.3.13.9 pandas.Series.dt.microsecond

`Series.dt.microsecond`

The microseconds of the datetime

#### 34.3.13.10 pandas.Series.dt.nanosecond

`Series.dt.nanosecond`

The nanoseconds of the datetime

#### 34.3.13.11 pandas.Series.dt.week

`Series.dt.week`

The week ordinal of the year

#### 34.3.13.12 pandas.Series.dt.weekofyear

`Series.dt.weekofyear`

The week ordinal of the year

#### 34.3.13.13 pandas.Series.dt.dayofweek

`Series.dt.dayofweek`

The day of the week with Monday=0, Sunday=6

#### 34.3.13.14 pandas.Series.dt.weekday

`Series.dt.weekday`

The day of the week with Monday=0, Sunday=6

#### 34.3.13.15 pandas.Series.dt.dayofyear

`Series.dt.dayofyear`

The ordinal day of the year

#### 34.3.13.16 pandas.Series.dt.quarter

`Series.dt.quarter`

The quarter of the date

#### 34.3.13.17 pandas.Series.dt.is\_month\_start

`Series.dt.is_month_start`

Logical indicating if first day of month (defined by frequency)

#### 34.3.13.18 pandas.Series.dt.is\_month\_end

`Series.dt.is_month_end`

Indicator for whether the date is the last day of the month.

**Returns** Series or array

For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

**See also:**

[`is\_month\_start`](#) Indicator for whether the date is the first day of the month.

### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> dates
0    2018-02-27
1    2018-02-28
2    2018-03-01
dtype: datetime64[ns]
>>> dates.dt.is_month_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_end
array([False,  True, False], dtype=bool)
```

### 34.3.13.19 pandas.Series.dt.is\_quarter\_start

#### Series.dt.is\_quarter\_start

Indicator for whether the date is the first day of a quarter.

**Returns** `is_quarter_start` : Series or DatetimeIndex

The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

`quarter` Return the quarter of the date.

`is_quarter_end` Similar property for indicating the quarter start.

#### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                                           periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_start=df.dates.dt.is_quarter_start)
   dates      quarter  is_quarter_start
0 2017-03-30         1                False
1 2017-03-31         1                False
2 2017-04-01         2                 True
3 2017-04-02         2                False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_start
array([False, False,  True, False])
```

### 34.3.13.20 pandas.Series.dt.is\_quarter\_end

#### Series.dt.is\_quarter\_end

Indicator for whether the date is the last day of a quarter.

**Returns** `is_quarter_end` : Series or DatetimeIndex

The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

`quarter` Return the quarter of the date.

`is_quarter_start` Similar property indicating the quarter start.

## Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                                           periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_end=df.dates.dt.is_quarter_end)
   dates      quarter  is_quarter_end
0 2017-03-30         1             False
1 2017-03-31         1              True
2 2017-04-01         2             False
3 2017-04-02         2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_end
array([False,  True, False, False])
```

### 34.3.13.21 pandas.Series.dt.is\_year\_start

`Series.dt.is_year_start`

Indicate whether the date is the first day of a year.

**Returns** Series or DatetimeIndex

The same type as the original data with boolean values. Series will have the same name and index. `DatetimeIndex` will have the same name.

**See also:**

[`is\_year\_end`](#) Similar property indicating the last day of the year.

## Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0    2017-12-30
1    2017-12-31
2    2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2     True
dtype: bool
```



```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_start
array([False, False,  True])
```

### 34.3.13.22 pandas.Series.dt.is\_year\_end

**Series.dt.is\_year\_end**

Indicate whether the date is the last day of the year.

**Returns Series or DatetimeIndex**

The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

[\*is\\_year\\_start\*](#) Similar property indicating the start of the year.

#### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on DatetimeIndex.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0    2017-12-30
1    2017-12-31
2    2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_end
array([False,  True, False])
```

### 34.3.13.23 pandas.Series.dt.is\_leap\_year

**Series.dt.is\_leap\_year**

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

**Returns Series or ndarray**

Booleans indicating if dates belong to a leap year.

**Examples**

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False], dtype=bool)
```

```
>>> dates = pd.Series(idx)
>>> dates_series
0    2012-12-31
1    2013-12-31
2    2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0      True
1     False
2     False
dtype: bool
```

**34.3.13.24 pandas.Series.dt.daysinmonth**

`Series.dt.daysinmonth`

The number of days in the month

**34.3.13.25 pandas.Series.dt.days\_in\_month**

`Series.dt.days_in_month`

The number of days in the month

**34.3.13.26 pandas.Series.dt.tz**

`Series.dt.tz`

**34.3.13.27 pandas.Series.dt.freq**

`Series.dt.freq`

**Datetime Methods**

---

*Series.dt.to\_period(\*args, \*\*kwargs)*

Cast to `PeriodIndex` at a particular frequency.

Continued on next page

Table 39 – continued from previous page

<code>Series.dt.to_pydatetime()</code>	Return the data as an array of native Python datetime objects
<code>Series.dt.tz_localize(*args, **kwargs)</code>	Localize tz-naive DatetimeIndex to tz-aware DateTimeIndex.
<code>Series.dt.tz_convert(*args, **kwargs)</code>	Convert tz-aware DatetimeIndex from one time zone to another.
<code>Series.dt.normalize(*args, **kwargs)</code>	Convert times to midnight.
<code>Series.dt.strftime(*args, **kwargs)</code>	Convert to Index using specified date_format.
<code>Series.dt.round(*args, **kwargs)</code>	round the data to the specified <i>freq</i> .
<code>Series.dt.floor(*args, **kwargs)</code>	floor the data to the specified <i>freq</i> .
<code>Series.dt.ceil(*args, **kwargs)</code>	ceil the data to the specified <i>freq</i> .
<code>Series.dt.month_name(*args, **kwargs)</code>	Return the month names of the DateTimeIndex with specified locale.
<code>Series.dt.day_name(*args, **kwargs)</code>	Return the day names of the DateTimeIndex with specified locale.

### 34.3.13.28 pandas.Series.dt.to\_period

`Series.dt.to_period(*args, **kwargs)`

Cast to PeriodIndex at a particular frequency.

Converts DatetimeIndex to PeriodIndex.

**Parameters** *freq* : string or Offset, optional

One of pandas' *offset strings* or an Offset object. Will be inferred by default.

**Returns**

**PeriodIndex**

**Raises** **ValueError**

When converting a DatetimeIndex with non-regular values, so that a frequency cannot be inferred.

**See also:**

`pandas.PeriodIndex` Immutable ndarray holding ordinal values

`pandas.DatetimeIndex.to_pydatetime` Return DatetimeIndex as object

### Examples

```
>>> df = pd.DataFrame({"y": [1,2,3]},
...                     index=pd.to_datetime(["2000-03-31 00:00:00",
...                                             "2000-05-31 00:00:00",
...                                             "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]', freq='M')
```

Infer the daily frequency

```
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]', freq='D')
```

### 34.3.13.29 pandas.Series.dt.to\_pydatetime

`Series.dt.to_pydatetime()`

Return the data as an array of native Python datetime objects

Timezone information is retained if present.

**Warning:** Python's datetime uses microsecond resolution, which is lower than pandas (nanosecond). The values are truncated.

**Returns** `numpy.ndarray`

object dtype array containing native Python datetime objects.

**See also:**

`datetime.datetime` Standard library value for a datetime.

#### Examples

```
>>> s = pd.Series(pd.date_range('20180310', periods=2))
>>> s
0    2018-03-10
1    2018-03-11
dtype: datetime64[ns]
```

```
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 11, 0, 0)], dtype=object)
```

pandas' nanosecond precision is truncated to microseconds.

```
>>> s = pd.Series(pd.date_range('20180310', periods=2, freq='ns'))
>>> s
0    2018-03-10 00:00:00.000000000
1    2018-03-10 00:00:00.000000001
dtype: datetime64[ns]
```

```
>>> s.dt.to_pydatetime()
array([datetime.datetime(2018, 3, 10, 0, 0),
       datetime.datetime(2018, 3, 10, 0, 0)], dtype=object)
```

### 34.3.13.30 pandas.Series.dt.tz\_localize

`Series.dt.tz_localize(*args, **kwargs)`

Localize tz-naive DatetimeIndex to tz-aware DatetimeIndex.

This method takes a time zone (tz) naive DatetimeIndex object and makes this time zone aware. It does not move the time to another time zone. Time zone localization helps to switch from time zone aware to time zone unaware objects.

**Parameters** `tz` : string, pytz.timezone, dateutil.tz.tzfile or None

Time zone to convert timestamps to. Passing None will remove the time zone information preserving local time.

**ambiguous** : str { 'infer', 'NaT', 'raise' } or bool array, default 'raise'

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

**errors** : { 'raise', 'coerce' }, default 'raise'

- 'raise' will raise a **NonExistentTimeError** if a timestamp is not valid in the specified time zone (e.g. due to a transition from or to DST time)
- 'coerce' will return NaT if the timestamp can not be converted to the specified time zone

New in version 0.19.0.

**Returns** DatetimeIndex

Index converted to the specified time zone.

**Raises** TypeError

If the DatetimeIndex is tz-aware and tz is not None.

**See also:**

**`DatetimeIndex.tz_convert`** Convert tz-aware DatetimeIndex from one time zone to another.

## Examples

```
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
              '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Localize DatetimeIndex in US/Eastern time zone:

```
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
              '2018-03-02 09:00:00-05:00',
              '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

With the `tz=None`, we can remove the time zone information while keeping the local time (not converted to UTC):

```
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
              '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

### 34.3.13.31 pandas.Series.dt.tz\_convert

`Series.dt.tz_convert(*args, **kwargs)`

Convert tz-aware DatetimeIndex from one time zone to another.

**Parameters** `tz`: string, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time. Corresponding timestamps would be converted to this time zone of the DatetimeIndex. A `tz` of None will convert to UTC and remove the timezone information.

**Returns**

**normalized** [DatetimeIndex]

**Raises** `TypeError`

If DatetimeIndex is tz-naive.

**See also:**

[`DatetimeIndex.tz`](#) A timezone that has a variable offset from UTC

[`DatetimeIndex.tz\_localize`](#) Localize tz-naive DatetimeIndex to a given time zone, or remove time-zone from a tz-aware DatetimeIndex.

### Examples

With the `tz` parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.DatetimeIndex(start='2014-08-01 09:00',
...                          freq='H', periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
              '2014-08-01 10:00:00+02:00',
              '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
              '2014-08-01 03:00:00-05:00',
              '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.DatetimeIndex(start='2014-08-01 09:00', freq='H',
...                          periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
              dtype='datetime64[ns]', freq='H')
```

### 34.3.13.32 pandas.Series.dt.normalize

`Series.dt.normalize(*args, **kwargs)`

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

#### Returns DatetimeIndex or Series

The same type as the original data. Series will have the same name and index. `DatetimeIndex` will have the same name.

See also:

**floor** Floor the datetimes to the specified freq.

**ceil** Ceil the datetimes to the specified freq.

**round** Round the datetimes to the specified freq.

### Examples

```
>>> idx = pd.DatetimeIndex(start='2014-08-01 10:00', freq='H',
...                        periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
              dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
              dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

### 34.3.13.33 pandas.Series.dt.strftime

`Series.dt.strftime(*args, **kwargs)`

Convert to Index using specified `date_format`.

Return an Index of formatted strings specified by `date_format`, which supports the same string format as the python standard library. Details of the string format can be found in [python string format doc](#)

**Parameters** `date_format` : str

Date format string (e.g. “%Y-%m-%d”).

**Returns** Index

Index of formatted strings

**See also:**

[`pandas.to\_datetime`](#) Convert the given argument to datetime

[`DatetimeIndex.normalize`](#) Return DatetimeIndex with times to midnight.

[`DatetimeIndex.round`](#) Round the DatetimeIndex to the specified freq.

[`DatetimeIndex.floor`](#) Floor the DatetimeIndex to the specified freq.

## Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                      periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

### 34.3.13.34 pandas.Series.dt.round

`Series.dt.round(*args, **kwargs)`  
round the data to the specified *freq*.

**Parameters** `freq` : str or Offset

The frequency level to round the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** DatetimeIndex, TimedeltaIndex, or Series

Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

**ValueError** if the ‘freq’ cannot be converted.

## Examples

### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
              '2018-01-01 12:01:00'],
             dtype='datetime64[ns]', freq='T')
```

(continues on next page)



(continued from previous page)

```
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
              '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

**Series**

```
>>> pd.Series(rng).dt.round("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

**34.3.13.35 pandas.Series.dt.floor**

`Series.dt.floor(*args, **kwargs)`  
 floor the data to the specified *freq*.

**Parameters** *freq* : str or Offset

The frequency level to floor the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** `DatetimeIndex`, `TimedeltaIndex`, or `Series`

Index of the same type for a `DatetimeIndex` or `TimedeltaIndex`, or a `Series` with the same index for a `Series`.

**Raises**

**ValueError** if the ‘freq’ cannot be converted.

**Examples****DatetimeIndex**

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
              '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
              '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

**Series**

```
>>> pd.Series(rng).dt.floor("H")
0    2018-01-01 11:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

### 34.3.13.36 pandas.Series.dt.ceil

`Series.dt.ceil(*args, **kwargs)`  
ceil the data to the specified *freq*.

**Parameters** `freq` : str or Offset

The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See *frequency aliases* for a list of possible *freq* values.

**Returns** `DatetimeIndex, TimedeltaIndex, or Series`

Index of the same type for a `DatetimeIndex` or `TimedeltaIndex`, or a `Series` with the same index for a `Series`.

**Raises**

**ValueError** if the ‘freq’ cannot be converted.

#### Examples

##### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
              '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
              '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

##### Series

```
>>> pd.Series(rng).dt.ceil("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 13:00:00
dtype: datetime64[ns]
```

### 34.3.13.37 pandas.Series.dt.month\_name

`Series.dt.month_name(*args, **kwargs)`

Return the month names of the `DateTimeIndex` with specified locale.

**Parameters** `locale` : string, default None (English locale)

locale determining the language in which to return the month name

**Returns** `month_names` : Index

Index of month names

.. versionadded:: 0.23.0

**34.3.13.38 pandas.Series.dt.day\_name**`Series.dt.day_name(*args, **kwargs)`

Return the day names of the DateTimeIndex with specified locale.

**Parameters** `locale` : string, default None (English locale)

locale determining the language in which to return the day name

**Returns** `month_names` : Index

Index of day names

**.. versionadded:: 0.23.0****Timedelta Properties**

<code>Series.dt.days</code>	Number of days for each element.
<code>Series.dt.seconds</code>	Number of seconds ( $\geq 0$ and less than 1 day) for each element.
<code>Series.dt.microseconds</code>	Number of microseconds ( $\geq 0$ and less than 1 second) for each element.
<code>Series.dt.nanoseconds</code>	Number of nanoseconds ( $\geq 0$ and less than 1 microsecond) for each element.
<code>Series.dt.components</code>	Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

**34.3.13.39 pandas.Series.dt.days**`Series.dt.days`

Number of days for each element.

**34.3.13.40 pandas.Series.dt.seconds**`Series.dt.seconds`Number of seconds ( $\geq 0$  and less than 1 day) for each element.**34.3.13.41 pandas.Series.dt.microseconds**`Series.dt.microseconds`Number of microseconds ( $\geq 0$  and less than 1 second) for each element.**34.3.13.42 pandas.Series.dt.nanoseconds**`Series.dt.nanoseconds`Number of nanoseconds ( $\geq 0$  and less than 1 microsecond) for each element.**34.3.13.43 pandas.Series.dt.components**`Series.dt.components`

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanosec-

onds) of the Timedeltas.

**Returns**

**a DataFrame**

### Timedelta Methods

<code>Series.dt.to_pytimedelta()</code>	Return an array of native <i>datetime.timedelta</i> objects.
<code>Series.dt.total_seconds(*args, **kwargs)</code>	Return total duration of each element expressed in seconds.

#### 34.3.13.44 pandas.Series.dt.to\_pytimedelta

`Series.dt.to_pytimedelta()`

Return an array of native *datetime.timedelta* objects.

Python's standard *datetime* library uses a different representation *timedelta*'s. This method converts a Series of pandas Timedeltas to *datetime.timedelta* format with the same length as the original Series.

**Returns** **a** : `numpy.ndarray`

1D array containing data with *datetime.timedelta* type.

**See also:**

`datetime.timedelta`

### Examples

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0    0 days
1    1 days
2    2 days
3    3 days
4    4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.to_pytimedelta()
array([datetime.timedelta(0), datetime.timedelta(1),
       datetime.timedelta(2), datetime.timedelta(3),
       datetime.timedelta(4)], dtype=object)
```

#### 34.3.13.45 pandas.Series.dt.total\_seconds

`Series.dt.total_seconds(*args, **kwargs)`

Return total duration of each element expressed in seconds.

This method is available directly on *TimedeltaIndex* and on Series containing *timedelta* values under the `.dt` namespace.

**Returns** **seconds** : `Float64Index` or `Series`

When the calling object is a `TimedeltaIndex`, the return type is a `Float64Index`. When the calling object is a `Series`, the return type is `Series` of type `float64` whose index is the same as the original.

See also:

`datetime.timedelta.total_seconds` Standard library version of this method.

`TimedeltaIndex.components` Return a `DataFrame` with components of each `Timedelta`.

## Examples

### Series

```
>>> s = pd.Series(pd.to_timedelta(np.arange(5), unit='d'))
>>> s
0    0 days
1    1 days
2    2 days
3    3 days
4    4 days
dtype: timedelta64[ns]
```

```
>>> s.dt.total_seconds()
0         0.0
1    86400.0
2   172800.0
3   259200.0
4   345600.0
dtype: float64
```

### TimedeltaIndex

```
>>> idx = pd.to_timedelta(np.arange(5), unit='d')
>>> idx
TimedeltaIndex(['0 days', '1 days', '2 days', '3 days', '4 days'],
               dtype='timedelta64[ns]', freq=None)
```

```
>>> idx.total_seconds()
Float64Index([0.0, 86400.0, 172800.0, 259200.000000000003, 345600.0],
             dtype='float64')
```

## 34.3.14 String handling

`Series.str` can be used to access the values of the series as strings and apply several methods to it. These can be accessed like `Series.str.<function/property>`.

<code>Series.str.capitalize()</code>	Convert strings in the <code>Series/Index</code> to be capitalized.
<code>Series.str.cat([others, sep, na_rep, join])</code>	Concatenate strings in the <code>Series/Index</code> with given separator.
<code>Series.str.center(width[, fillchar])</code>	Filling left and right side of strings in the <code>Series/Index</code> with an additional character.

Continued on next page

Table 42 – continued from previous page

<code>Series.str.contains(pat[, case, flags, na, ...])</code>	Test if pattern or regex is contained within a string of a Series or Index.
<code>Series.str.count(pat[, flags])</code>	Count occurrences of pattern in each string of the Series/Index.
<code>Series.str.decode(encoding[, errors])</code>	Decode character string in the Series/Index using indicated encoding.
<code>Series.str.encode(encoding[, errors])</code>	Encode character string in the Series/Index using indicated encoding.
<code>Series.str.endswith(pat[, na])</code>	Test if the end of each string element matches a pattern.
<code>Series.str.extract(pat[, flags, expand])</code>	For each subject string in the Series, extract groups from the first match of regular expression pat.
<code>Series.str.extractall(pat[, flags])</code>	For each subject string in the Series, extract groups from all matches of regular expression pat.
<code>Series.str.find(sub[, start, end])</code>	Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].
<code>Series.str.findall(pat[, flags])</code>	Find all occurrences of pattern or regular expression in the Series/Index.
<code>Series.str.get(i)</code>	Extract element from each component at specified position.
<code>Series.str.index(sub[, start, end])</code>	Return lowest indexes in each strings where the substring is fully contained between [start:end].
<code>Series.str.join(sep)</code>	Join lists contained as elements in the Series/Index with passed delimiter.
<code>Series.str.len()</code>	Compute length of each string in the Series/Index.
<code>Series.str.ljust(width[, fillchar])</code>	Filling right side of strings in the Series/Index with an additional character.
<code>Series.str.lower()</code>	Convert strings in the Series/Index to lowercase.
<code>Series.str.lstrip([to_strip])</code>	Strip whitespace (including newlines) from each string in the Series/Index from left side.
<code>Series.str.match(pat[, case, flags, na, ...])</code>	Determine if each string matches a regular expression.
<code>Series.str.normalize(form)</code>	Return the Unicode normal form for the strings in the Series/Index.
<code>Series.str.pad(width[, side, fillchar])</code>	Pad strings in the Series/Index with an additional character to specified side.
<code>Series.str.partition([pat, expand])</code>	Split the string at the first occurrence of <i>sep</i> , and return 3 elements containing the part before the separator, the separator itself, and the part after the separator.
<code>Series.str.repeat(repeats)</code>	Duplicate each string in the Series/Index by indicated number of times.
<code>Series.str.replace(pat, repl[, n, case, ...])</code>	Replace occurrences of pattern/regex in the Series/Index with some other string.
<code>Series.str.rfind(sub[, start, end])</code>	Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end].
<code>Series.str.rindex(sub[, start, end])</code>	Return highest indexes in each strings where the substring is fully contained between [start:end].
<code>Series.str.rjust(width[, fillchar])</code>	Filling left side of strings in the Series/Index with an additional character.

Continued on next page

Table 42 – continued from previous page

<code>Series.str.rpartition([pat, expand])</code>	Split the string at the last occurrence of <i>sep</i> , and return 3 elements containing the part before the separator, the separator itself, and the part after the separator.
<code>Series.str.rstrip([to_strip])</code>	Strip whitespace (including newlines) from each string in the Series/Index from right side.
<code>Series.str.slice([start, stop, step])</code>	Slice substrings from each element in the Series/Index
<code>Series.str.slice_replace([start, stop, repl])</code>	Replace a positional slice of a string with another value.
<code>Series.str.split([pat, n, expand])</code>	Split strings around given separator/delimiter.
<code>Series.str.rsplit([pat, n, expand])</code>	Split each string in the Series/Index by the given delimiter string, starting at the end of the string and working to the front.
<code>Series.str.startswith(pat[, na])</code>	Test if the start of each string element matches a pattern.
<code>Series.str.strip([to_strip])</code>	Strip whitespace (including newlines) from each string in the Series/Index from left and right sides.
<code>Series.str.swapcase()</code>	Convert strings in the Series/Index to be swapcased.
<code>Series.str.title()</code>	Convert strings in the Series/Index to titlecase.
<code>Series.str.translate(table[, deletechars])</code>	Map all characters in the string through the given mapping table.
<code>Series.str.upper()</code>	Convert strings in the Series/Index to uppercase.
<code>Series.str.wrap(width, **kwargs)</code>	Wrap long strings in the Series/Index to be formatted in paragraphs with length less than a given width.
<code>Series.str.zfill(width)</code>	Filling left side of strings in the Series/Index with 0.
<code>Series.str.isalnum()</code>	Check whether all characters in each string in the Series/Index are alphanumeric.
<code>Series.str.isalpha()</code>	Check whether all characters in each string in the Series/Index are alphabetic.
<code>Series.str.isdigit()</code>	Check whether all characters in each string in the Series/Index are digits.
<code>Series.str.isspace()</code>	Check whether all characters in each string in the Series/Index are whitespace.
<code>Series.str.islower()</code>	Check whether all characters in each string in the Series/Index are lowercase.
<code>Series.str.isupper()</code>	Check whether all characters in each string in the Series/Index are uppercase.
<code>Series.str.istitle()</code>	Check whether all characters in each string in the Series/Index are titlecase.
<code>Series.str.isnumeric()</code>	Check whether all characters in each string in the Series/Index are numeric.
<code>Series.str.isdecimal()</code>	Check whether all characters in each string in the Series/Index are decimal.
<code>Series.str.get_dummies([sep])</code>	Split each string in the Series by sep and return a frame of dummy/indicator variables.

### 34.3.14.1 pandas.Series.str.capitalize

`Series.str.capitalize()`

Convert strings in the Series/Index to be capitalized.

Equivalent to `str.capitalize()`.

**Returns**

**Series/Index of objects**

**See also:**

***Series.str.lower*** Converts all characters to lowercase.

***Series.str.upper*** Converts all characters to uppercase.

***Series.str.title*** Converts first character of each word to uppercase and remaining to lowercase.

***Series.str.capitalize*** Converts first character to uppercase and remaining to lowercase.

***Series.str.swapcase*** Converts uppercase to lowercase and lowercase to uppercase.

**Examples**

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2    this is a sentence
3          SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0          lower
1        capitals
2    this is a sentence
3          swapcase
dtype: object
```

```
>>> s.str.upper()
0          LOWER
1        CAPITALS
2    THIS IS A SENTENCE
3          SWAPCASE
dtype: object
```

```
>>> s.str.title()
0          Lower
1        Capitals
2    This Is A Sentence
3          Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0          Lower
1        Capitals
2    This is a sentence
3          Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0          LOWER
1        capitals
2    THIS IS A SENTENCE
3          sWaPcAsE
dtype: object
```



### 34.3.14.2 pandas.Series.str.cat

`Series.str.cat` (*others=None, sep=None, na\_rep=None, join=None*)

Concatenate strings in the Series/Index with given separator.

If *others* is specified, this function concatenates the Series/Index and elements of *others* element-wise. If *others* is not passed, then all values in the Series/Index are concatenated into a single string with a given *sep*.

**Parameters** *others* : Series, Index, DataFrame, np.ndarray or list-like

Series, Index, DataFrame, np.ndarray (one- or two-dimensional) and other list-likes of strings must have the same length as the calling Series/Index, with the exception of indexed objects (i.e. Series/Index/DataFrame) if *join* is not None.

If *others* is a list-like that contains a combination of Series, np.ndarray (1-dim) or list-like, then all elements will be unpacked and must satisfy the above criteria individually.

If *others* is None, the method returns the concatenation of all strings in the calling Series/Index.

**sep** : string or None, default None

If None, concatenates without any separator.

**na\_rep** : string or None, default None

Representation that is inserted for all missing values:

- If *na\_rep* is None, and *others* is None, missing values in the Series/Index are omitted from the result.
- If *na\_rep* is None, and *others* is not None, a row containing a missing value in any of the columns (before concatenation) will have a missing value in the result.

**join** : { 'left', 'right', 'outer', 'inner' }, default None

Determines the join-style between the calling Series/Index and any Series/Index/DataFrame in *others* (objects without an index need to match the length of the calling Series/Index). If None, alignment is disabled, but this option will be removed in a future version of pandas and replaced with a default of 'left'. To disable alignment, use *.values* on any Series/Index/DataFrame in *others*.

New in version 0.23.0.

**Returns** *concat* : str or Series/Index of objects

If *others* is None, *str* is returned, otherwise a *Series/Index* (same type as caller) of objects is returned.

See also:

[\*split\*](#) Split each string in the Series/Index

### Examples

When not passing *others*, all values are concatenated into a single string:

```
>>> s = pd.Series(['a', 'b', np.nan, 'd'])
>>> s.str.cat(sep=' ')
'a b d'
```

By default, NA values in the Series are ignored. Using *na\_rep*, they can be given a representation:

```
>>> s.str.cat(sep=' ', na_rep='?')
'a b ? d'
```

If *others* is specified, corresponding values are concatenated with the separator. Result will be a Series of strings.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',')
0    a,A
1    b,B
2    NaN
3    d,D
dtype: object
```

Missing values will remain missing in the result, but can again be represented using *na\_rep*

```
>>> s.str.cat(['A', 'B', 'C', 'D'], sep=',', na_rep='-')
0    a,A
1    b,B
2    -,C
3    d,D
dtype: object
```

If *sep* is not specified, the values are concatenated without separation.

```
>>> s.str.cat(['A', 'B', 'C', 'D'], na_rep='-')
0    aA
1    bB
2    -C
3    dD
dtype: object
```

Series with different indexes can be aligned before concatenation. The *join*-keyword works as in other methods.

```
>>> t = pd.Series(['d', 'a', 'e', 'c'], index=[3, 0, 4, 2])
>>> s.str.cat(t, join=None, na_rep='-')
0    ad
1    ba
2    -e
3    dc
dtype: object
>>>
>>> s.str.cat(t, join='left', na_rep='-')
0    aa
1    b-
2    -c
3    dd
dtype: object
>>>
>>> s.str.cat(t, join='outer', na_rep='-')
0    aa
1    b-
2    -c
```

(continues on next page)

(continued from previous page)

```

3    dd
4    -e
dtype: object
>>>
>>> s.str.cat(t, join='inner', na_rep='-')
0    aa
2    -c
3    dd
dtype: object
>>>
>>> s.str.cat(t, join='right', na_rep='-')
3    dd
0    aa
4    -e
2    -c
dtype: object

```

For more examples, see [here](#).

#### 34.3.14.3 pandas.Series.str.center

`Series.str.center` (*width*, *fillchar*=`' '`)

Filling left and right side of strings in the Series/Index with an additional character. Equivalent to `str.center()`.

**Parameters** *width* : int

Minimum width of resulting string; additional characters will be filled with *fillchar*

**fillchar** : str

Additional character for filling, default is whitespace

**Returns**

**filled** [Series/Index of objects]

#### 34.3.14.4 pandas.Series.str.contains

`Series.str.contains` (*pat*, *case*=`True`, *flags*=0, *na*=`nan`, *regex*=`True`)

Test if pattern or regex is contained within a string of a Series or Index.

Return boolean Series or Index based on whether a given pattern or regex is contained within a string of a Series or Index.

**Parameters** *pat* : str

Character sequence or regular expression.

**case** : bool, default True

If True, case sensitive.

**flags** : int, default 0 (no flags)

Flags to pass through to the re module, e.g. `re.IGNORECASE`.

**na** : default NaN

Fill value for missing values.

**regex** : bool, default True

If True, assumes the pat is a regular expression.

If False, treats the pat as a literal string.

**Returns Series or Index of boolean values**

A Series or Index of boolean values indicating whether the given pattern is contained within the string of each element of the Series or Index.

**See also:**

*match* analogous, but stricter, relying on re.match instead of re.search

## Examples

Returning a Series of booleans using only a literal pattern.

```
>>> s1 = pd.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
>>> s1.str.contains('og', regex=False)
0    False
1     True
2    False
3    False
4      NaN
dtype: object
```

Returning an Index of booleans using only a literal pattern.

```
>>> ind = pd.Index(['Mouse', 'dog', 'house and parrot', '23.0', np.NaN])
>>> ind.str.contains('23', regex=False)
Index([False, False, False, True, nan], dtype='object')
```

Specifying case sensitivity using *case*.

```
>>> s1.str.contains('oG', case=True, regex=True)
0    False
1    False
2    False
3    False
4      NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN* replaces NaN values with *False*. If Series or Index does not contain NaN values the resultant dtype will be *bool*, otherwise, an *object* dtype.

```
>>> s1.str.contains('og', na=False, regex=True)
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

Returning 'house' and 'parrot' within same string.

```
>>> s1.str.contains('house|parrot', regex=True)
0    False
1    False
2     True
3    False
4      NaN
dtype: object
```

Ignoring case sensitivity using *flags* with regex.

```
>>> import re
>>> s1.str.contains('PARROT', flags=re.IGNORECASE, regex=True)
0    False
1    False
2     True
3    False
4      NaN
dtype: object
```

Returning any digit using regular expression.

```
>>> s1.str.contains('\d', regex=True)
0    False
1    False
2    False
3     True
4      NaN
dtype: object
```

Ensure *pat* is not a literal pattern when *regex* is set to *True*. Note in the following example one might expect only *s2[1]* and *s2[3]* to return *True*. However, *'0'* as a regex matches any character followed by a 0.

```
>>> s2 = pd.Series(['40', '40.0', '41', '41.0', '35'])
>>> s2.str.contains('.0', regex=True)
0     True
1     True
2    False
3     True
4    False
dtype: bool
```

#### 34.3.14.5 pandas.Series.str.count

`Series.str.count` (*pat*, *flags=0*, *\*\*kwargs*)

Count occurrences of pattern in each string of the Series/Index.

This function is used to count the number of times a particular regex pattern is repeated in each of the string elements of the *Series*.

**Parameters** *pat* : str

Valid regular expression.

**flags** : int, default 0, meaning no flags

Flags for the *re* module. For a complete list, [see here](#).

**\*\*kwargs**

For compatability with other string methods. Not used.

**Returns** **counts** : Series or Index

Same type as the calling object containing the integer counts.

**See also:**

**re** Standard library module for regular expressions.

**str.count** Standard library version, without regular expression support.

## Notes

Some characters need to be escaped when passing in *pat*. eg. '\$' has a special meaning in regex and must be escaped when finding this literal character.

## Examples

```
>>> s = pd.Series(['A', 'B', 'Aaba', 'Baca', np.nan, 'CABA', 'cat'])
>>> s.str.count('a')
0      0.0
1      0.0
2      2.0
3      2.0
4      NaN
5      0.0
6      1.0
dtype: float64
```

Escape '\$' to find the literal dollar sign.

```
>>> s = pd.Series(['$', 'B', 'Aab$', '$$ca', 'C$B$', 'cat'])
>>> s.str.count('\$')
0      1
1      0
2      1
3      2
4      2
5      0
dtype: int64
```

This is also available on Index

```
>>> pd.Index(['A', 'A', 'Aaba', 'cat']).str.count('a')
Int64Index([0, 0, 2, 1], dtype='int64')
```

### 34.3.14.6 pandas.Series.str.decode

**Series.str.decode** (*encoding*, *errors*='strict')

Decode character string in the Series/Index using indicated encoding. Equivalent to `str.decode()` in python2 and `bytes.decode()` in python3.

**Parameters**

**encoding** [str]

**errors** [str, optional]

**Returns**

**decoded** [Series/Index of objects]

### 34.3.14.7 pandas.Series.str.encode

`Series.str.encode(encoding, errors='strict')`

Encode character string in the Series/Index using indicated encoding. Equivalent to `str.encode()`.

**Parameters**

**encoding** [str]

**errors** [str, optional]

**Returns**

**encoded** [Series/Index of objects]

### 34.3.14.8 pandas.Series.str.endswith

`Series.str.endswith(pat, na=nan)`

Test if the end of each string element matches a pattern.

Equivalent to `str.endswith()`.

**Parameters** **pat** : str

Character sequence. Regular expressions are not accepted.

**na** : object, default NaN

Object shown if element tested is not a string.

**Returns** **Series or Index of bool**

A Series of booleans indicating whether the given pattern matches the end of each string element.

**See also:**

**str.endswith** Python standard library string method.

**Series.str.startswith** Same as `endswith`, but tests the start of string.

**Series.str.contains** Tests if string element contains a pattern.

### Examples

```
>>> s = pd.Series(['bat', 'bear', 'caT', np.nan])
>>> s
0    bat
1   bear
2   caT
3   NaN
dtype: object
```

```
>>> s.str.endswith('t')
0      True
1     False
2     False
3       NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN*.

```
>>> s.str.endswith('t', na=False)
0      True
1     False
2     False
3     False
dtype: bool
```

### 34.3.14.9 pandas.Series.str.extract

`Series.str.extract` (*pat, flags=0, expand=True*)

For each subject string in the Series, extract groups from the first match of regular expression *pat*.

**Parameters** *pat* : string

Regular expression pattern with capturing groups

**flags** : int, default 0 (no flags)

re module flags, e.g. `re.IGNORECASE`

**expand** : bool, default True

- If True, return DataFrame.
- If False, return Series/Index/DataFrame.

New in version 0.18.0.

**Returns**

**DataFrame with one row for each subject string, and one column for each group. Any capture group names in regular expression *pat* will be used for column names; otherwise capture group numbers will be used. The dtype of each result column is always object, even when no match is found. If *expand=False* and *pat* has only one capture group, then return a Series (if subject is a Series) or Index (if subject is an Index).**

See also:

[`extractall`](#) returns all matches (not just the first match)

### Examples

A pattern with two groups will return a DataFrame with two columns. Non-matches will be NaN.



```
>>> s = Series(['a1', 'b2', 'c3'])
>>> s.str.extract(r'([ab])(\d)')
   0  1
0   a  1
1   b  2
2  NaN NaN
```

A pattern may contain optional groups.

```
>>> s.str.extract(r'([ab])?(\d)')
   0  1
0   a  1
1   b  2
2  NaN 3
```

Named groups will become column names in the result.

```
>>> s.str.extract(r'(?P<letter>[ab])(?P<digit>\d)')
   letter digit
0      a      1
1      b      2
2    NaN    NaN
```

A pattern with one group will return a DataFrame with one column if `expand=True`.

```
>>> s.str.extract(r'[ab](\d)', expand=True)
   0
0   1
1   2
2  NaN
```

A pattern with one group will return a Series if `expand=False`.

```
>>> s.str.extract(r'[ab](\d)', expand=False)
0      1
1      2
2    NaN
dtype: object
```

#### 34.3.14.10 pandas.Series.str.extractall

`Series.str.extractall(pat, flags=0)`

For each subject string in the Series, extract groups from all matches of regular expression `pat`. When each subject string in the Series has exactly one match, `extractall(pat).xs(0, level='match')` is the same as `extract(pat)`.

New in version 0.18.0.

**Parameters** `pat` : string

Regular expression pattern with capturing groups

`flags` : int, default 0 (no flags)

re module flags, e.g. `re.IGNORECASE`

**Returns**

A DataFrame with one row for each match, and one column for each

group. Its rows have a `MultiIndex` with first levels that come from the subject Series. The last level is named `'match'` and indicates the order in the subject. Any capture group names in regular expression `pat` will be used for column names; otherwise capture group numbers will be used.

See also:

`extract` returns first match only (not all matches)

## Examples

A pattern with one group will return a DataFrame with one column. Indices with no matches will not appear in the result.

```
>>> s = Series(["a1a2", "b1", "c1"], index=["A", "B", "C"])
>>> s.str.extractall(r"[ab](\d)")
      match
A 0      1
  1      2
B 0      1
```

Capture group names are used for column names of the result.

```
>>> s.str.extractall(r"[ab](?P<digit>\d)")
      digit
      match
A 0      1
  1      2
B 0      1
```

A pattern with two groups will return a DataFrame with two columns.

```
>>> s.str.extractall(r"(?P<letter>[ab])(?P<digit>\d)")
      letter digit
      match
A 0      a      1
  1      a      2
B 0      b      1
```

Optional groups that do not match are NaN in the result.

```
>>> s.str.extractall(r"(?P<letter>[ab])?(?P<digit>\d)")
      letter digit
      match
A 0      a      1
  1      a      2
B 0      b      1
C 0     NaN      1
```

### 34.3.14.11 pandas.Series.str.find

`Series.str.find(sub, start=0, end=None)`

Return lowest indexes in each strings in the Series/Index where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard `str.find()`.

**Parameters** `sub` : str

Substring being searched

`start` : int

Left edge index

`end` : int

Right edge index

**Returns**

**found** [Series/Index of integer values]

**See also:**

[`rfind`](#) Return highest indexes in each strings

### 34.3.14.12 pandas.Series.str.findall

`Series.str.findall(pat, flags=0, **kwargs)`

Find all occurrences of pattern or regular expression in the Series/Index.

Equivalent to applying `re.findall()` to all the elements in the Series/Index.

**Parameters** `pat` : string

Pattern or regular expression.

`flags` : int, default 0

re module flags, e.g. `re.IGNORECASE` (default is 0, which means no flags).

**Returns** Series/Index of lists of strings

All non-overlapping matches of pattern or regular expression in each string of this Series/Index.

**See also:**

[`count`](#) Count occurrences of pattern or regular expression in each string of the Series/Index.

[`extractall`](#) For each string in the Series, extract groups from all matches of regular expression and return a DataFrame with one row for each match and one column for each group.

[`re.findall`](#) The equivalent re function to all non-overlapping matches of pattern or regular expression in string, as a list of strings.

### Examples

```
>>> s = pd.Series(['Lion', 'Monkey', 'Rabbit'])
```

The search for the pattern 'Monkey' returns one match:

```
>>> s.str.findall('Monkey')
0      []
1    [Monkey]
2      []
dtype: object
```

On the other hand, the search for the pattern 'MONKEY' doesn't return any match:

```
>>> s.str.findall('MONKEY')
0      []
1      []
2      []
dtype: object
```

Flags can be added to the pattern or regular expression. For instance, to find the pattern 'MONKEY' ignoring the case:

```
>>> import re
>>> s.str.findall('MONKEY', flags=re.IGNORECASE)
0      []
1    [Monkey]
2      []
dtype: object
```

When the pattern matches more than one string in the Series, all matches are returned:

```
>>> s.str.findall('on')
0    [on]
1    [on]
2      []
dtype: object
```

Regular expressions are supported too. For instance, the search for all the strings ending with the word 'on' is shown next:

```
>>> s.str.findall('on$')
0    [on]
1      []
2      []
dtype: object
```

If the pattern is found more than once in the same string, then a list of multiple strings is returned:

```
>>> s.str.findall('b')
0      []
1      []
2    [b, b]
dtype: object
```

#### 34.3.14.13 pandas.Series.str.get

`Series.str.get(i)`

Extract element from each component at specified position.

Extract element from lists, tuples, or strings in each element in the Series/Index.

**Parameters** `i`: int

Position of element to extract.

### Returns

**items** [Series/Index of objects]

### Examples

```
>>> s = pd.Series(["String",
                  (1, 2, 3),
                  ["a", "b", "c"],
                  123, -456,
                  {1:"Hello", "2":"World"}])
>>> s
0          String
1          (1, 2, 3)
2          [a, b, c]
3           123
4          -456
5    {1: 'Hello', '2': 'World'}
dtype: object
```

```
>>> s.str.get(1)
0          t
1          2
2          b
3         NaN
4         NaN
5       Hello
dtype: object
```

```
>>> s.str.get(-1)
0          g
1          3
2          c
3         NaN
4         NaN
5         NaN
dtype: object
```

#### 34.3.14.14 pandas.Series.str.index

`Series.str.index` (*sub*, *start=0*, *end=None*)

Return lowest indexes in each strings where the substring is fully contained between [start:end]. This is the same as `str.find` except instead of returning -1, it raises a `ValueError` when the substring is not found. Equivalent to standard `str.index`.

**Parameters** *sub* : str

Substring being searched

**start** : int

Left edge index

**end** : int

Right edge index

#### Returns

**found** [Series/Index of objects]

**See also:**

[\*rinter\*](#) Return highest indexes in each strings

### 34.3.14.15 pandas.Series.str.join

`Series.str.join(sep)`

Join lists contained as elements in the Series/Index with passed delimiter.

If the elements of a Series are lists themselves, join the content of these lists using the delimiter passed to the function. This function is an equivalent to `str.join()`.

**Parameters** `sep` : str

Delimiter to use between list entries.

#### Returns

**Series/Index:** object

**See also:**

[\*str.join\*](#) Standard library version of this method.

[\*Series.str.split\*](#) Split strings around given separator/delimiter.

#### Notes

If any of the lists does not contain string objects the result of the join will be *NaN*.

#### Examples

Example with a list that contains non-string elements.

```
>>> s = pd.Series(['lion', 'elephant', 'zebra'],
...               [1.1, 2.2, 3.3],
...               ['cat', np.nan, 'dog'],
...               ['cow', 4.5, 'goat'],
...               ['duck', ['swan', 'fish'], 'guppy'])
>>> s
0      [lion, elephant, zebra]
1      [1.1, 2.2, 3.3]
2      [cat, nan, dog]
3      [cow, 4.5, goat]
4      [duck, [swan, fish], guppy]
dtype: object
```

Join all lists using an '-', the lists containing object(s) of types other than str will become a NaN.

```
>>> s.str.join('-')
0    lion-elephant-zebra
1                      NaN
2                      NaN
3                      NaN
4                      NaN
dtype: object
```

#### 34.3.14.16 pandas.Series.str.len

`Series.str.len()`

Compute length of each string in the Series/Index.

##### Returns

**lengths** [Series/Index of integer values]

#### 34.3.14.17 pandas.Series.str.ljust

`Series.str.ljust(width, fillchar=' ')`

Filling right side of strings in the Series/Index with an additional character. Equivalent to `str.ljust()`.

##### Parameters

**width** : int

Minimum width of resulting string; additional characters will be filled with `fillchar`

##### **fillchar** : str

Additional character for filling, default is whitespace

##### Returns

**filled** [Series/Index of objects]

#### 34.3.14.18 pandas.Series.str.lower

`Series.str.lower()`

Convert strings in the Series/Index to lowercase.

Equivalent to `str.lower()`.

##### Returns

**Series/Index of objects**

See also:

**`Series.str.lower`** Converts all characters to lowercase.

**`Series.str.upper`** Converts all characters to uppercase.

**`Series.str.title`** Converts first character of each word to uppercase and remaining to lowercase.

**`Series.str.capitalize`** Converts first character to uppercase and remaining to lowercase.

**`Series.str.swapcase`** Converts uppercase to lowercase and lowercase to uppercase.

## Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2  this is a sentence
3        SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0          lower
1        capitals
2  this is a sentence
3        swapcase
dtype: object
```

```
>>> s.str.upper()
0          LOWER
1        CAPITALS
2  THIS IS A SENTENCE
3        SWAPCASE
dtype: object
```

```
>>> s.str.title()
0          Lower
1        Capitals
2  This Is A Sentence
3        Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0          Lower
1        Capitals
2  This is a sentence
3        Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0          LOWER
1        capitals
2  THIS IS A SENTENCE
3        sWaPcAsE
dtype: object
```

### 34.3.14.19 pandas.Series.str.lstrip

`Series.str.lstrip` (*to\_strip=None*)

Strip whitespace (including newlines) from each string in the Series/Index from left side. Equivalent to `str.lstrip()`.

#### Returns

**stripped** [Series/Index of objects]



#### 34.3.14.20 pandas.Series.str.match

`Series.str.match(pat, case=True, flags=0, na=nan, as_indexer=None)`

Determine if each string matches a regular expression.

**Parameters** `pat` : string

Character sequence or regular expression

**case** : boolean, default True

If True, case sensitive

**flags** : int, default 0 (no flags)

re module flags, e.g. re.IGNORECASE

**na** [default NaN, fill value for missing values.]

**as\_indexer**

Deprecated since version 0.21.0.

**Returns**

Series/array of boolean values

**See also:**

[`contains`](#) analogous, but less strict, relying on re.search instead of re.match

[`extract`](#) extract matched groups

#### 34.3.14.21 pandas.Series.str.normalize

`Series.str.normalize(form)`

Return the Unicode normal form for the strings in the Series/Index. For more information on the forms, see the `unicodedata.normalize()`.

**Parameters** `form` : {'NFC', 'NFKC', 'NFD', 'NFKD'}

Unicode form

**Returns**

**normalized** [Series/Index of objects]

#### 34.3.14.22 pandas.Series.str.pad

`Series.str.pad(width, side='left', fillchar=' ')`

Pad strings in the Series/Index with an additional character to specified side.

**Parameters** `width` : int

Minimum width of resulting string; additional characters will be filled with spaces

**side** [{'left', 'right', 'both'}, default 'left']

**fillchar** : str

Additional character for filling, default is whitespace

**Returns****padded** [Series/Index of objects]**34.3.14.23 pandas.Series.str.partition**`Series.str.partition` (*pat*=' ', *expand*=True)

Split the string at the first occurrence of *sep*, and return 3 elements containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 elements containing the string itself, followed by two empty strings.

**Parameters** *pat* : string, default whitespace

String to split on.

**expand** : bool, default True

- If True, return DataFrame/MultiIndex expanding dimensionality.
- If False, return Series/Index.

**Returns****split** [DataFrame/MultiIndex or Series/Index of objects]**See also:**[`rpartition`](#) Split the string at the last occurrence of *sep***Examples**

```
>>> s = Series(['A_B_C', 'D_E_F', 'X'])
0    A_B_C
1    D_E_F
2         X
dtype: object
```

```
>>> s.str.partition('_')
   0  1  2
0  A  _  B_C
1  D  _  E_F
2  X
```

```
>>> s.str.rpartition('_')
   0  1  2
0  A_B  _  C
1  D_E  _  F
2         X
```

**34.3.14.24 pandas.Series.str.repeat**`Series.str.repeat` (*repeats*)

Duplicate each string in the Series/Index by indicated number of times.

**Parameters** *repeats* : int or array

Same value for all (int) or different value per (array)

**Returns****repeated** [Series/Index of objects]**34.3.14.25 pandas.Series.str.replace**`Series.str.replace(pat, repl, n=-1, case=None, flags=0, regex=True)`

Replace occurrences of pattern/regex in the Series/Index with some other string. Equivalent to `str.replace()` or `re.sub()`.

**Parameters** `pat` : string or compiled regex

String can be a character sequence or regular expression.

New in version 0.20.0: `pat` also accepts a compiled regex.**repl** : string or callable

Replacement string or a callable. The callable is passed the regex match object and must return a replacement string to be used. See `re.sub()`.

New in version 0.20.0: `repl` also accepts a callable.**n** : int, default -1 (all)

Number of replacements to make from start

**case** : boolean, default None

- If True, case sensitive (the default if `pat` is a string)
- Set to False for case insensitive
- Cannot be set if `pat` is a compiled regex

**flags** : int, default 0 (no flags)

- re module flags, e.g. `re.IGNORECASE`
- Cannot be set if `pat` is a compiled regex

**regex** : boolean, default True

- If True, assumes the passed-in pattern is a regular expression.
- If False, treats the pattern as a literal string
- Cannot be set to False if `pat` is a compiled regex or `repl` is a callable.

New in version 0.23.0.

**Returns****replaced** [Series/Index of objects]**Raises** `ValueError`

- if `regex` is False and `repl` is a callable or `pat` is a compiled regex
- if `pat` is a compiled regex and `case` or `flags` is set

**Notes**

When `pat` is a compiled regex, all flags should be included in the compiled regex. Use of `case`, `flags`, or `regex=False` with a compiled regex will raise an error.

## Examples

When *pat* is a string and *regex* is True (the default), the given *pat* is compiled as a regex. When *repl* is a string, it replaces matching regex patterns as with `re.sub()`. NaN value(s) in the Series are left as is:

```
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f.', 'ba', regex=True)
0    bao
1    baz
2    NaN
dtype: object
```

When *pat* is a string and *regex* is False, every *pat* is replaced with *repl* as with `str.replace()`:

```
>>> pd.Series(['f.o', 'fuz', np.nan]).str.replace('f.', 'ba', regex=False)
0    bao
1    fuz
2    NaN
dtype: object
```

When *repl* is a callable, it is called on every *pat* using `re.sub()`. The callable should expect one positional argument (a regex object) and return a string.

To get the idea:

```
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace('f', repr)
0    <_sre.SRE_Match object; span=(0, 1), match='f'>oo
1    <_sre.SRE_Match object; span=(0, 1), match='f'>uz
2    NaN
dtype: object
```

Reverse every lowercase alphabetic word:

```
>>> repl = lambda m: m.group(0)[::-1]
>>> pd.Series(['foo 123', 'bar baz', np.nan]).str.replace(r'[a-z]+', repl)
0    oof 123
1    rab zab
2    NaN
dtype: object
```

Using regex groups (extract second group and swap case):

```
>>> pat = r"(?P<one>\w+) (?P<two>\w+) (?P<three>\w+)"
>>> repl = lambda m: m.group('two').swapcase()
>>> pd.Series(['One Two Three', 'Foo Bar Baz']).str.replace(pat, repl)
0    tWO
1    bAR
dtype: object
```

Using a compiled regex with flags

```
>>> regex_pat = re.compile(r'FUZ', flags=re.IGNORECASE)
>>> pd.Series(['foo', 'fuz', np.nan]).str.replace(regex_pat, 'bar')
0    foo
1    bar
2    NaN
dtype: object
```

#### 34.3.14.26 pandas.Series.str.rfind

`Series.str.rfind(sub, start=0, end=None)`

Return highest indexes in each strings in the Series/Index where the substring is fully contained between [start:end]. Return -1 on failure. Equivalent to standard `str.rfind()`.

**Parameters** `sub` : str

Substring being searched

`start` : int

Left edge index

`end` : int

Right edge index

**Returns**

**found** [Series/Index of integer values]

**See also:**

[`find`](#) Return lowest indexes in each strings

#### 34.3.14.27 pandas.Series.str.rindex

`Series.str.rindex(sub, start=0, end=None)`

Return highest indexes in each strings where the substring is fully contained between [start:end]. This is the same as `str.rfind` except instead of returning -1, it raises a `ValueError` when the substring is not found. Equivalent to standard `str.rindex`.

**Parameters** `sub` : str

Substring being searched

`start` : int

Left edge index

`end` : int

Right edge index

**Returns**

**found** [Series/Index of objects]

**See also:**

[`index`](#) Return lowest indexes in each strings

#### 34.3.14.28 pandas.Series.str.rjust

`Series.str.rjust(width, fillchar='')`

Filling left side of strings in the Series/Index with an additional character. Equivalent to `str.rjust()`.

**Parameters** `width` : int

Minimum width of resulting string; additional characters will be filled with `fillchar`

**fillchar** : str

Additional character for filling, default is whitespace

**Returns**

**filled** [Series/Index of objects]

### 34.3.14.29 pandas.Series.str.rpartition

`Series.str.rpartition(pat=' ', expand=True)`

Split the string at the last occurrence of *sep*, and return 3 elements containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return 3 elements containing two empty strings, followed by the string itself.

**Parameters** **pat** : string, default whitespace

String to split on.

**expand** : bool, default True

- If True, return DataFrame/MultiIndex expanding dimensionality.
- If False, return Series/Index.

**Returns**

**split** [DataFrame/MultiIndex or Series/Index of objects]

**See also:**

[\*partition\*](#) Split the string at the first occurrence of *sep*

### Examples

```
>>> s = Series(['A_B_C', 'D_E_F', 'X'])
0    A_B_C
1    D_E_F
2         X
dtype: object
```

```
>>> s.str.partition('_')
   0  1  2
0  A  _  B_C
1  D  _  E_F
2  X
```

```
>>> s.str.rpartition('_')
   0  1  2
0  A_B  _  C
1  D_E  _  F
2         X
```

#### 34.3.14.30 pandas.Series.str.rstrip

`Series.str.rstrip` (*to\_strip=None*)

Strip whitespace (including newlines) from each string in the Series/Index from right side. Equivalent to `str.rstrip()`.

**Returns**

**stripped** [Series/Index of objects]

#### 34.3.14.31 pandas.Series.str.slice

`Series.str.slice` (*start=None, stop=None, step=None*)

Slice substrings from each element in the Series/Index

**Parameters**

**start** [int or None]

**stop** [int or None]

**step** [int or None]

**Returns**

**sliced** [Series/Index of objects]

#### 34.3.14.32 pandas.Series.str.slice\_replace

`Series.str.slice_replace` (*start=None, stop=None, repl=None*)

Replace a positional slice of a string with another value.

**Parameters** **start** : int, optional

Left index position to use for the slice. If not specified (None), the slice is unbounded on the left, i.e. slice from the start of the string.

**stop** : int, optional

Right index position to use for the slice. If not specified (None), the slice is unbounded on the right, i.e. slice until the end of the string.

**repl** : str, optional

String for replacement. If not specified (None), the sliced region is replaced with an empty string.

**Returns** **replaced** : Series or Index

Same type as the original object.

**See also:**

[`Series.str.slice`](#) Just slicing without replacement.

#### Examples

```
>>> s = pd.Series(['a', 'ab', 'abc', 'abdc', 'abcde'])
>>> s
0      a
1     ab
2    abc
3   abdc
4  abcde
dtype: object
```

Specify just *start*, meaning replace *start* until the end of the string with *repl*.

```
>>> s.str.slice_replace(1, repl='X')
0    aX
1    aX
2    aX
3    aX
4    aX
dtype: object
```

Specify just *stop*, meaning the start of the string to *stop* is replaced with *repl*, and the rest of the string is included.

```
>>> s.str.slice_replace(stop=2, repl='X')
0      X
1      X
2     Xc
3    Xdc
4   Xcde
dtype: object
```

Specify *start* and *stop*, meaning the slice from *start* to *stop* is replaced with *repl*. Everything before or after *start* and *stop* is included as is.

```
>>> s.str.slice_replace(start=1, stop=3, repl='X')
0      aX
1      aX
2      aX
3     aXc
4    aXde
dtype: object
```

### 34.3.14.33 pandas.Series.str.split

`Series.str.split` (*pat=None, n=-1, expand=False*)

Split strings around given separator/delimiter.

Split each string in the caller's values by given pattern, propagating NaN values. Equivalent to `str.split()`.

**Parameters** *pat* : str, optional

String or regular expression to split on. If not specified, split on whitespace.

**n** : int, default -1 (all)

Limit number of splits in output. None, 0 and -1 will be interpreted as return all splits.

**expand** : bool, default False

Expand the splitted strings into separate columns.



- If `True`, return `DataFrame`/`MultiIndex` expanding dimensionality.
- If `False`, return `Series`/`Index`, containing lists of strings.

**Returns Series, Index, DataFrame or MultiIndex**

Type matches caller unless `expand=True` (see Notes).

**See also:**

`str.split` Standard library version of this method.

`Series.str.get_dummies` Split each string into dummy variables.

`Series.str.partition` Split string on a separator, returning the before, separator, and after components.

**Notes**

The handling of the `n` keyword depends on the number of found splits:

- If found splits  $> n$ , make first  $n$  splits only
- If found splits  $\leq n$ , make all splits
- If for a certain row the number of found splits  $< n$ , append `None` for padding up to  $n$  if `expand=True`

If using `expand=True`, `Series` and `Index` callers return `DataFrame` and `MultiIndex` objects, respectively.

**Examples**

```
>>> s = pd.Series(["this is good text", "but this is even better"])
```

By default, `split` will return an object of the same size having lists containing the split elements

```
>>> s.str.split()
0      [this, is, good, text]
1      [but, this, is, even, better]
dtype: object
>>> s.str.split("random")
0      [this is good text]
1      [but this is even better]
dtype: object
```

When using `expand=True`, the split elements will expand out into separate columns.

For `Series` object, output return type is `DataFrame`.

```
>>> s.str.split(expand=True)
   0      1      2      3      4
0  this   is  good  text  None
1  but   this   is  even  better
>>> s.str.split(" is ", expand=True)
   0      1
0   this   good text
1  but this   even better
```

For `Index` object, output return type is `MultiIndex`.

```
>>> i = pd.Index(["ba 100 001", "ba 101 002", "ba 102 003"])
>>> i.str.split(expand=True)
MultiIndex(levels=[['ba'], ['100', '101', '102'], ['001', '002', '003']],
            labels=[[0, 0, 0], [0, 1, 2], [0, 1, 2]])
```

Parameter *n* can be used to limit the number of splits in the output.

```
>>> s.str.split("is", n=1)
0      [th,  is good text]
1    [but th,  is even better]
dtype: object
>>> s.str.split("is", n=1, expand=True)
      0      1
0    th    is good text
1  but th  is even better
```

If NaN is present, it is propagated throughout the columns during the split.

```
>>> s = pd.Series(["this is good text", "but this is even better", np.nan])
>>> s.str.split(n=3, expand=True)
      0      1      2      3
0  this    is  good    text
1  but   this    is  even better
2   NaN   NaN   NaN    NaN
```

#### 34.3.14.34 pandas.Series.str.rsplit

`Series.str.rsplit` (*pat=None, n=-1, expand=False*)

Split each string in the Series/Index by the given delimiter string, starting at the end of the string and working to the front. Equivalent to `str.rsplit()`.

**Parameters** *pat* : string, default None

Separator to split on. If None, splits on whitespace

*n* : int, default -1 (all)

None, 0 and -1 will be interpreted as return all splits

**expand** : bool, default False

- If True, return DataFrame/MultiIndex expanding dimensionality.
- If False, return Series/Index.

**Returns**

`split` [Series/Index or DataFrame/MultiIndex of objects]

#### 34.3.14.35 pandas.Series.str.startswith

`Series.str.startswith` (*pat, na=nan*)

Test if the start of each string element matches a pattern.

Equivalent to `str.startswith()`.

**Parameters** *pat* : str

Character sequence. Regular expressions are not accepted.

**na** : object, default NaN

Object shown if element tested is not a string.

#### Returns Series or Index of bool

A Series of booleans indicating whether the given pattern matches the start of each string element.

#### See also:

**str.startswith** Python standard library string method.

**Series.str.endswith** Same as startswith, but tests the end of string.

**Series.str.contains** Tests if string element contains a pattern.

#### Examples

```
>>> s = pd.Series(['bat', 'Bear', 'cat', np.nan])
>>> s
0    bat
1   Bear
2    cat
3   NaN
dtype: object
```

```
>>> s.str.startswith('b')
0    True
1   False
2   False
3   NaN
dtype: object
```

Specifying *na* to be *False* instead of *NaN*.

```
>>> s.str.startswith('b', na=False)
0    True
1   False
2   False
3   False
dtype: bool
```

#### 34.3.14.36 pandas.Series.str.strip

**Series.str.strip** (*to\_strip=None*)

Strip whitespace (including newlines) from each string in the Series/Index from left and right sides. Equivalent to `str.strip()`.

#### Returns

**stripped** [Series/Index of objects]

### 34.3.14.37 pandas.Series.str.swapcase

`Series.str.swapcase()`

Convert strings in the Series/Index to be swapcased.

Equivalent to `str.swapcase()`.

#### Returns

Series/Index of objects

See also:

**`Series.str.lower`** Converts all characters to lowercase.

**`Series.str.upper`** Converts all characters to uppercase.

**`Series.str.title`** Converts first character of each word to uppercase and remaining to lowercase.

**`Series.str.capitalize`** Converts first character to uppercase and remaining to lowercase.

**`Series.str.swapcase`** Converts uppercase to lowercase and lowercase to uppercase.

#### Examples

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2  this is a sentence
3        SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0          lower
1        capitals
2  this is a sentence
3        swapcase
dtype: object
```

```
>>> s.str.upper()
0          LOWER
1        CAPITALS
2  THIS IS A SENTENCE
3        SWAPCASE
dtype: object
```

```
>>> s.str.title()
0          Lower
1        Capitals
2  This Is A Sentence
3        Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0          Lower
1        Capitals
```

(continues on next page)

(continued from previous page)

```

2    This is a sentence
3                Swapcase
dtype: object

```

```

>>> s.str.swapcase()
0                LOWER
1            capitals
2    THIS IS A SENTENCE
3            sWaPcAsE
dtype: object

```

#### 34.3.14.38 pandas.Series.str.title

`Series.str.title()`

Convert strings in the Series/Index to titlecase.

Equivalent to `str.title()`.

##### Returns

**Series/Index of objects**

**See also:**

**`Series.str.lower`** Converts all characters to lowercase.

**`Series.str.upper`** Converts all characters to uppercase.

**`Series.str.title`** Converts first character of each word to uppercase and remaining to lowercase.

**`Series.str.capitalize`** Converts first character to uppercase and remaining to lowercase.

**`Series.str.swapcase`** Converts uppercase to lowercase and lowercase to uppercase.

##### Examples

```

>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0                lower
1            CAPITALS
2    this is a sentence
3            SwApCaSe
dtype: object

```

```

>>> s.str.lower()
0                lower
1            capitals
2    this is a sentence
3            swapcase
dtype: object

```

```

>>> s.str.upper()
0                LOWER
1            CAPITALS
2    THIS IS A SENTENCE

```

(continues on next page)

(continued from previous page)

```
3          SWAPCASE
dtype: object
```

```
>>> s.str.title()
0          Lower
1        Capitals
2  This Is A Sentence
3        Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0          Lower
1        Capitals
2  This is a sentence
3        Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0          LOWER
1        capitals
2  THIS IS A SENTENCE
3        sWaPcAsE
dtype: object
```

### 34.3.14.39 pandas.Series.str.translate

`Series.str.translate(table, deletechars=None)`

Map all characters in the string through the given mapping table. Equivalent to standard `str.translate()`. Note that the optional argument `deletechars` is only valid if you are using python 2. For python 3, character deletion should be specified via the `table` argument.

**Parameters** `table` : dict (python 3), str or None (python 2)

In python 3, `table` is a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted. `str.maketrans()` is a helper function for making translation tables. In python 2, `table` is either a string of length 256 or None. If the `table` argument is None, no translation is applied and the operation simply removes the characters in `deletechars`. `string.maketrans()` is a helper function for making translation tables.

**deletechars** : str, optional (python 2)

A string of characters to delete. This argument is only valid in python 2.

**Returns**

**translated** [Series/Index of objects]

### 34.3.14.40 pandas.Series.str.upper

`Series.str.upper()`

Convert strings in the Series/Index to uppercase.

Equivalent to `str.upper()`.

**Returns****Series/Index of objects****See also:***Series.str.lower* Converts all characters to lowercase.*Series.str.upper* Converts all characters to uppercase.*Series.str.title* Converts first character of each word to uppercase and remaining to lowercase.*Series.str.capitalize* Converts first character to uppercase and remaining to lowercase.*Series.str.swapcase* Converts uppercase to lowercase and lowercase to uppercase.**Examples**

```
>>> s = pd.Series(['lower', 'CAPITALS', 'this is a sentence', 'SwApCaSe'])
>>> s
0          lower
1        CAPITALS
2    this is a sentence
3          SwApCaSe
dtype: object
```

```
>>> s.str.lower()
0          lower
1        capitals
2    this is a sentence
3          swapcase
dtype: object
```

```
>>> s.str.upper()
0          LOWER
1        CAPITALS
2    THIS IS A SENTENCE
3          SWAPCASE
dtype: object
```

```
>>> s.str.title()
0          Lower
1        Capitals
2    This Is A Sentence
3          Swapcase
dtype: object
```

```
>>> s.str.capitalize()
0          Lower
1        Capitals
2    This is a sentence
3          Swapcase
dtype: object
```

```
>>> s.str.swapcase()
0          LOWER
```

(continues on next page)

(continued from previous page)

```
1          capitals
2  THIS IS A SENTENCE
3          sWaPcAsE
dtype: object
```

#### 34.3.14.41 pandas.Series.str.wrap

`Series.str.wrap` (*width*, *\*\*kwargs*)

Wrap long strings in the Series/Index to be formatted in paragraphs with length less than a given width.

This method has the same keyword parameters and defaults as `textwrap.TextWrapper`.

**Parameters** `width` : int

Maximum line-width

**expand\_tabs** : bool, optional

If true, tab characters will be expanded to spaces (default: True)

**replace\_whitespace** : bool, optional

If true, each whitespace character (as defined by `string.whitespace`) remaining after tab expansion will be replaced by a single space (default: True)

**drop\_whitespace** : bool, optional

If true, whitespace that, after wrapping, happens to end up at the beginning or end of a line is dropped (default: True)

**break\_long\_words** : bool, optional

If true, then words longer than width will be broken in order to ensure that no lines are longer than width. If it is false, long words will not be broken, and some lines may be longer than width. (default: True)

**break\_on\_hyphens** : bool, optional

If true, wrapping will occur preferably on whitespace and right after hyphens in compound words, as it is customary in English. If false, only whitespaces will be considered as potentially good places for line breaks, but you need to set `break_long_words` to false if you want truly insecable words. (default: True)

**Returns**

**wrapped** [Series/Index of objects]

#### Notes

Internally, this method uses a `textwrap.TextWrapper` instance with default settings. To achieve behavior matching R's `stringr` library `str_wrap` function, use the arguments:

- `expand_tabs = False`
- `replace_whitespace = True`
- `drop_whitespace = True`
- `break_long_words = False`
- `break_on_hyphens = False`



## Examples

```
>>> s = pd.Series(['line to be wrapped', 'another line to be wrapped'])
>>> s.str.wrap(12)
0          line to be\nwrapped
1  another line\nto be\nwrapped
```

### 34.3.14.42 pandas.Series.str.zfill

`Series.str.zfill(width)`

Filling left side of strings in the Series/Index with 0. Equivalent to `str.zfill()`.

**Parameters** `width`: int

Minimum width of resulting string; additional characters will be filled with 0

**Returns**

**filled** [Series/Index of objects]

### 34.3.14.43 pandas.Series.str.isalnum

`Series.str.isalnum()`

Check whether all characters in each string in the Series/Index are alphanumeric. Equivalent to `str.isalnum()`.

**Returns**

**is** [Series/array of boolean values]

### 34.3.14.44 pandas.Series.str.isalpha

`Series.str.isalpha()`

Check whether all characters in each string in the Series/Index are alphabetic. Equivalent to `str.isalpha()`.

**Returns**

**is** [Series/array of boolean values]

### 34.3.14.45 pandas.Series.str.isdigit

`Series.str.isdigit()`

Check whether all characters in each string in the Series/Index are digits. Equivalent to `str.isdigit()`.

**Returns**

**is** [Series/array of boolean values]

### 34.3.14.46 pandas.Series.str.isspace

`Series.str.isspace()`

Check whether all characters in each string in the Series/Index are whitespace. Equivalent to `str.isspace()`.

**Returns**

**is** [Series/array of boolean values]

#### **34.3.14.47 pandas.Series.str.islower**

`Series.str.islower()`

Check whether all characters in each string in the Series/Index are lowercase. Equivalent to `str.islower()`.

##### **Returns**

**is** [Series/array of boolean values]

#### **34.3.14.48 pandas.Series.str.isupper**

`Series.str.isupper()`

Check whether all characters in each string in the Series/Index are uppercase. Equivalent to `str.isupper()`.

##### **Returns**

**is** [Series/array of boolean values]

#### **34.3.14.49 pandas.Series.str.istitle**

`Series.str.istitle()`

Check whether all characters in each string in the Series/Index are titlecase. Equivalent to `str.istitle()`.

##### **Returns**

**is** [Series/array of boolean values]

#### **34.3.14.50 pandas.Series.str.isnumeric**

`Series.str.isnumeric()`

Check whether all characters in each string in the Series/Index are numeric. Equivalent to `str.isnumeric()`.

##### **Returns**

**is** [Series/array of boolean values]

#### **34.3.14.51 pandas.Series.str.isdecimal**

`Series.str.isdecimal()`

Check whether all characters in each string in the Series/Index are decimal. Equivalent to `str.isdecimal()`.

##### **Returns**

**is** [Series/array of boolean values]

#### **34.3.14.52 pandas.Series.str.get\_dummies**

`Series.str.get_dummies(sep='|')`

Split each string in the Series by sep and return a frame of dummy/indicator variables.

**Parameters** `sep`: string, default “|”

String to split on.

**Returns****dummies** [DataFrame]**See also:**`pandas.get_dummies`**Examples**

```
>>> Series(['a|b', 'a', 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  1  0  0
2  1  0  1
```

```
>>> Series(['a|b', np.nan, 'a|c']).str.get_dummies()
   a  b  c
0  1  1  0
1  0  0  0
2  1  0  1
```

### 34.3.15 Categorical

Pandas defines a custom data type for representing data that can take only a limited, fixed set of values. The dtype of a `Categorical` can be described by a `pandas.api.types.CategoricalDtype`.

---

<code>api.types.CategoricalDtype([categories, ordered])</code>	Type for categorical data with the categories and orderedness
--	---

---

#### 34.3.15.1 pandas.api.types.CategoricalDtype

**class** `pandas.api.types.CategoricalDtype` (*categories=None, ordered=None*)

Type for categorical data with the categories and orderedness

Changed in version 0.21.0.

**Parameters** **categories** : sequence, optional

Must be unique, and must not contain any nulls.

**ordered** [bool, default False]

**See also:**`pandas.Categorical`**Notes**

This class is useful for specifying the type of a `Categorical` independent of the values. See `CategoricalDtype` for more.

## Examples

```
>>> t = CategoricalDtype(categories=['b', 'a'], ordered=True)
>>> pd.Series(['a', 'b', 'a', 'c'], dtype=t)
0      a
1      b
2      a
3     NaN
dtype: category
Categories (2, object): [b < a]
```

## Attributes

<i>categories</i>	An Index containing the unique categories allowed.
<i>ordered</i>	Whether the categories have an ordered relationship

### pandas.api.types.CategoricalDtype.categories

`CategoricalDtype.categories`  
An Index containing the unique categories allowed.

### pandas.api.types.CategoricalDtype.ordered

`CategoricalDtype.ordered`  
Whether the categories have an ordered relationship

## Methods

None	
------	--

<i>api.types.CategoricalDtype.categories</i>	An Index containing the unique categories allowed.
<i>api.types.CategoricalDtype.ordered</i>	Whether the categories have an ordered relationship

Categorical data can be stored in a *pandas.Categorical*

<i>Categorical</i> (values[, categories, ordered, ...])	Represents a categorical variable in classic R / S-plus fashion
---	---

### 34.3.15.2 pandas.Categorical

**class** `pandas.Categorical` (*values, categories=None, ordered=None, dtype=None, fastpath=False*)  
Represents a categorical variable in classic R / S-plus fashion

*Categoricals* can only take on only a limited, and usually fixed, number of possible values (*categories*). In contrast to statistical categorical variables, a *Categorical* might have an order, but numerical operations (additions, divisions, ...) are not possible.

All values of the *Categorical* are either in *categories* or *np.nan*. Assigning values outside of *categories* will raise a *ValueError*. Order is defined by the order of the *categories*, not lexical order of the values.

**Parameters** **values** : list-like

The values of the categorical. If categories are given, values not in categories will be replaced with NaN.

**categories** : Index-like (unique), optional

The unique categories for this categorical. If not given, the categories are assumed to be the unique values of values.

**ordered** : boolean, (default False)

Whether or not this categorical is treated as a ordered categorical. If not given, the resulting categorical will not be ordered.

**dtype** : CategoricalDtype

An instance of *CategoricalDtype* to use for this categorical

New in version 0.21.0.

**Raises** **ValueError**

If the categories do not validate.

**TypeError**

If an explicit `ordered=True` is given but no *categories* and the *values* are not sortable.

**See also:**

*pandas.api.types.CategoricalDtype* Type for categorical data

*CategoricalIndex* An Index with an underlying *Categorical*

## Notes

See the [user guide](#) for more.

## Examples

```
>>> pd.Categorical([1, 2, 3, 1, 2, 3])
[1, 2, 3, 1, 2, 3]
Categories (3, int64): [1, 2, 3]
```

```
>>> pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
[a, b, c, a, b, c]
Categories (3, object): [a, b, c]
```

Ordered *Categoricals* can be sorted according to the custom order of the categories and can have a min and max value.

```

>>> c = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'], ordered=True,
...                     categories=['c', 'b', 'a'])
>>> c
[a, b, c, a, b, c]
Categories (3, object): [c < b < a]
>>> c.min()
'c'

```

## Attributes

<code>categories</code>	The categories of this categorical.
<code>codes</code>	The category codes of this categorical.
<code>ordered</code>	Whether the categories have an ordered relationship
<code>dtype</code>	The <code>CategoricalDtype</code> for this instance

## pandas.Categorical.categories

### `Categorical.categories`

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to `categories` is an inplace operation!

#### **Raises** `ValueError`

If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

#### **See also:**

`rename_categories`, `reorder_categories`, `add_categories`, `remove_categories`, `remove_unused_categories`, `set_categories`

## pandas.Categorical.codes

### `Categorical.codes`

The category codes of this categorical.

Level codes are an array of integer which are the positions of the real values in the categories array.

There is no setter, use the other categorical methods and the normal item setter to change values in the categorical.

## pandas.Categorical.ordered

### `Categorical.ordered`

Whether the categories have an ordered relationship

## pandas.Categorical.dtype

`Categorical.dtype`

The *CategoricalDtype* for this instance

## Methods

<code>from_codes(codes, categories[, ordered])</code>	Make a Categorical type from codes and categories arrays.
<code>__array__([dtype])</code>	The numpy array interface.

## pandas.Categorical.from\_codes

**classmethod** `Categorical.from_codes(codes, categories, ordered=False)`

Make a Categorical type from codes and categories arrays.

This constructor is useful if you already have codes and categories and so do not need the (computation intensive) factorization step, which is usually done on the constructor.

If your data does not follow this convention, please use the normal constructor.

**Parameters** `codes` : array-like, integers

An integer array, where each integer points to a category in categories or -1 for NaN

**categories** : index-like

The categories for the categorical. Items need to be unique.

**ordered** : boolean, (default False)

Whether or not this categorical is treated as a ordered categorical. If not given, the resulting categorical will be unordered.

## pandas.Categorical.\_\_array\_\_

`Categorical.__array__(dtype=None)`

The numpy array interface.

**Returns** `values` : numpy array

A numpy array of either the specified dtype or, if dtype==None (default), the same dtype as `categorical.categories.dtype`

The alternative `Categorical.from_codes()` constructor can be used when you have the categories and integer codes already:

<code>Categorical.from_codes(codes, categories[, ...])</code>	Make a Categorical type from codes and categories arrays.
---	---

The dtype information is available on the `Categorical`

<code>Categorical.dtype</code>	The <code>CategoricalDtype</code> for this instance
<code>Categorical.categories</code>	The categories of this categorical.
<code>Categorical.ordered</code>	Whether the categories have an ordered relationship
<code>Categorical.codes</code>	The category codes of this categorical.

`np.asarray(categorical)` works by implementing the array interface. Be aware, that this converts the Categorical back to a NumPy array, so categories and order information is not preserved!

<code>Categorical.__array__([dtype])</code>	The numpy array interface.
---	----------------------------

A Categorical can be stored in a Series or DataFrame. To create a Series of dtype category, use `cat = s.astype(dtype)` or `Series(..., dtype=dtype)` where dtype is either

- the string 'category'
- an instance of `CategoricalDtype`.

If the Series is of dtype CategoricalDtype, `Series.cat` can be used to change the categorical data. This accessor is similar to the `Series.dt` or `Series.str` and has the following usable methods and properties:

<code>Series.cat.categories</code>	The categories of this categorical.
<code>Series.cat.ordered</code>	Whether the categories have an ordered relationship
<code>Series.cat.codes</code>	

### 34.3.15.3 pandas.Series.cat.categories

`Series.cat.categories`

The categories of this categorical.

Setting assigns new values to each category (effectively a rename of each individual category).

The assigned value has to be a list-like object. All items must be unique and the number of items in the new categories must be the same as the number of items in the old categories.

Assigning to `categories` is a inplace operation!

#### Raises ValueError

If the new categories do not validate as categories or if the number of new categories is unequal the number of old categories

See also:

`rename_categories`, `reorder_categories`, `add_categories`, `remove_categories`, `remove_unused_categories`, `set_categories`

### 34.3.15.4 pandas.Series.cat.ordered

`Series.cat.ordered`

Whether the categories have an ordered relationship

### 34.3.15.5 pandas.Series.cat.codes

`Series.cat.codes`



<code>Series.cat.rename_categories(*args, **kwargs)</code>	Renames categories.
<code>Series.cat.reorder_categories(*args, **kwargs)</code>	Reorders categories as specified in new_categories.
<code>Series.cat.add_categories(*args, **kwargs)</code>	Add new categories.
<code>Series.cat.remove_categories(*args, **kwargs)</code>	Removes the specified categories.
<code>Series.cat.remove_unused_categories(*args, ...)</code>	Removes categories which are not used.
<code>Series.cat.set_categories(*args, **kwargs)</code>	Sets the categories to the specified new_categories.
<code>Series.cat.as_ordered(*args, **kwargs)</code>	Sets the Categorical to be ordered
<code>Series.cat.as_unordered(*args, **kwargs)</code>	Sets the Categorical to be unordered

### 34.3.15.6 pandas.Series.cat.rename\_categories

`Series.cat.rename_categories(*args, **kwargs)`

Renames categories.

**Parameters** `new_categories` : list-like, dict-like or callable

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.
- callable : a callable that is called on all items in the old categories and whose return values comprise the new categories.

New in version 0.21.0.

New in version 0.23.0.

**Warning:** Currently, Series are considered list like. In a future version of pandas they'll be considered dict-like.

**inplace** : boolean (default: False)

Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

**Returns** `cat` : Categorical or None

With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

**Raises** `ValueError`

If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

**See also:**

`reorder_categories`, `add_categories`, `remove_categories`,  
`remove_unused_categories`, `set_categories`

## Examples

```
>>> c = Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through

```
>>> c.rename_categories({'a': 'A', 'c': 'C'})
[A, A, b]
Categories (2, object): [A, b]
```

You may also provide a callable to create the new categories

```
>>> c.rename_categories(lambda x: x.upper())
[A, A, B]
Categories (2, object): [A, B]
```

### 34.3.15.7 pandas.Series.cat.reorder\_categories

`Series.cat.reorder_categories(*args, **kwargs)`

Reorders categories as specified in `new_categories`.

`new_categories` need to include all old categories and no new category items.

**Parameters** `new_categories` : Index-like

The categories in new order.

**ordered** : boolean, optional

Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

**inplace** : boolean (default: False)

Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

**cat** [Categorical with reordered categories or None if inplace.]

**Raises** `ValueError`

If the new categories do not contain all old category items or any new ones

**See also:**

`rename_categories`, `add_categories`, `remove_categories`,  
`remove_unused_categories`, `set_categories`

### 34.3.15.8 pandas.Series.cat.add\_categories

`Series.cat.add_categories(*args, **kwargs)`

Add new categories.

`new_categories` will be included at the last/highest place in the categories and will be unused directly after this call.

**Parameters** `new_categories` : category or list-like of category

The new categories to be included.

**inplace** : boolean (default: False)

Whether or not to add the categories inplace or return a copy of this categorical with added categories.

**Returns**

**cat** [Categorical with new categories added or None if inplace.]

**Raises** `ValueError`

If the new categories include old categories or do not validate as categories

**See also:**

`rename_categories`, `reorder_categories`, `remove_categories`,  
`remove_unused_categories`, `set_categories`

#### 34.3.15.9 `pandas.Series.cat.remove_categories`

`Series.cat.remove_categories(*args, **kwargs)`

Removes the specified categories.

*removals* must be included in the old categories. Values which were in the removed categories will be set to NaN

**Parameters** `removals` : category or list of categories

The categories which should be removed.

**inplace** : boolean (default: False)

Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

**Returns**

**cat** [Categorical with removed categories or None if inplace.]

**Raises** `ValueError`

If the removals are not contained in the categories

**See also:**

`rename_categories`, `reorder_categories`, `add_categories`,  
`remove_unused_categories`, `set_categories`

#### 34.3.15.10 `pandas.Series.cat.remove_unused_categories`

`Series.cat.remove_unused_categories(*args, **kwargs)`

Removes categories which are not used.

**Parameters** `inplace` : boolean (default: False)

Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

**Returns**

**cat** [Categorical with unused categories dropped or None if inplace.]

See also:

`rename_categories`, `reorder_categories`, `add_categories`, `remove_categories`,  
`set_categories`

### 34.3.15.11 pandas.Series.cat.set\_categories

`Series.cat.set_categories(*args, **kwargs)`

Sets the categories to the specified new\_categories.

*new\_categories* can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simply be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this method does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes on python3, which does not consider a S1 string equal to a single char python string.

**Parameters** *new\_categories* : Index-like

The categories in new order.

**ordered** : boolean, (default: False)

Whether or not the categorical is treated as an ordered categorical. If not given, do not change the ordered information.

**rename** : boolean (default: False)

Whether or not the new\_categories should be considered as a rename of the old categories or as reordered categories.

**inplace** : boolean (default: False)

Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

**cat** [Categorical with reordered categories or None if inplace.]

**Raises** `ValueError`

If *new\_categories* does not validate as categories

See also:

`rename_categories`, `reorder_categories`, `add_categories`, `remove_categories`,  
`remove_unused_categories`

### 34.3.15.12 pandas.Series.cat.as\_ordered

`Series.cat.as_ordered(*args, **kwargs)`

Sets the Categorical to be ordered

**Parameters** *inplace* : boolean (default: False)

Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to True

### 34.3.15.13 pandas.Series.cat.as\_unordered

`Series.cat.as_unordered(*args, **kwargs)`

Sets the Categorical to be unordered

**Parameters** `inplace` : boolean (default: False)

Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to False

### 34.3.16 Plotting

`Series.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `Series.plot.<kind>`.

<code>Series.plot([kind, ax, figsize, ...])</code>	Series plotting accessor and method
<code>Series.plot.area(**kwargs)</code>	Area plot
<code>Series.plot.bar(**kwargs)</code>	Vertical bar plot
<code>Series.plot.barh(**kwargs)</code>	Horizontal bar plot
<code>Series.plot.box(**kwargs)</code>	Boxplot
<code>Series.plot.density([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>Series.plot.hist([bins])</code>	Histogram
<code>Series.plot.kde([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>Series.plot.line(**kwargs)</code>	Line plot
<code>Series.plot.pie(**kwargs)</code>	Pie chart

#### 34.3.16.1 pandas.Series.plot.area

`Series.plot.area(**kwargs)`

Area plot

**Parameters** `**kwargs` : optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

`axes` [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

#### 34.3.16.2 pandas.Series.plot.bar

`Series.plot.bar(**kwargs)`

Vertical bar plot

**Parameters** `**kwargs` : optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

`axes` [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

### 34.3.16.3 pandas.Series.plot.barh

`Series.plot.barh(**kws)`

Horizontal bar plot

**Parameters** `**kws` : optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

`axes` [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

### 34.3.16.4 pandas.Series.plot.box

`Series.plot.box(**kws)`

Boxplot

**Parameters** `**kws` : optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

`axes` [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

### 34.3.16.5 pandas.Series.plot.density

`Series.plot.density(bw_method=None, ind=None, **kws)`

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, [kernel density estimation](#) (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters** `bw_method` : str, scalar or callable, optional

The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.

`ind` : NumPy array or integer, optional

Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If `ind` is a NumPy array, the KDE is evaluated at the points passed. If `ind` is an integer, `ind` number of equally spaced points are used.

`**kws` : optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

`axes` [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

**See also:**

`scipy.stats.gaussian_kde` Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

`DataFrame.plot.kde` Generate a KDE plot for a DataFrame.

## Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to overfitting, while using a large bandwidth value may result in underfitting:

```
>>> ax = s.plot.kde(bw_method=0.3)
```

```
>>> ax = s.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

### 34.3.16.6 pandas.Series.plot.hist

`Series.plot.hist` (*bins=10, \*\*kws*)

Histogram

**Parameters** **bins**: integer, default 10

Number of histogram bins to be used

**\*\*kws**: optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

**axes** [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

### 34.3.16.7 pandas.Series.plot.kde

`Series.plot.kde` (*bw\_method=None, ind=None, \*\*kws*)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, [kernel density estimation](#) (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters** **bw\_method**: str, scalar or callable, optional

The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.

**ind**: NumPy array or integer, optional

Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

**\*\*kws**: optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

#### Returns

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

#### See also:

**scipy.stats.gaussian\_kde** Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

**DataFrame.plot.kde** Generate a KDE plot for a DataFrame.

#### Examples

Given a Series of points randomly sampled from an unknown distribution, estimate its PDF using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> s = pd.Series([1, 2, 2.5, 3, 3.5, 4, 5])
>>> ax = s.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to overfitting, while using a large bandwidth value may result in underfitting:

```
>>> ax = s.plot.kde(bw_method=0.3)
```

```
>>> ax = s.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = s.plot.kde(ind=[1, 2, 3, 4, 5])
```

### 34.3.16.8 pandas.Series.plot.line

`Series.plot.line(**kws)`

Line plot

**Parameters** ‘\*\*kws’: optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

#### Returns

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

#### Examples

```
>>> s = pd.Series([1, 3, 2])
>>> s.plot.line()
```

### 34.3.16.9 pandas.Series.plot.pie

`Series.plot.pie(**kws)`

Pie chart



**Parameters** `**kwargs` : optional

Additional keyword arguments are documented in `pandas.Series.plot()`.

**Returns**

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

<code>Series.hist([by, ax, grid, xlabelsize, ...])</code>	Draw histogram of the input series using matplotlib
---	---

### 34.3.17 Serialization / IO / Conversion

<code>Series.to_pickle(path[, compression, protocol])</code>	Pickle (serialize) object to file.
<code>Series.to_csv([path, index, sep, na_rep, ...])</code>	Write Series to a comma-separated values (csv) file
<code>Series.to_dict([into])</code>	Convert Series to {label -> value} dict or dict-like object.
<code>Series.to_excel(excel_writer[, sheet_name, ...])</code>	Write Series to an excel sheet
<code>Series.to_frame([name])</code>	Convert Series to DataFrame
<code>Series.to_xarray()</code>	Return an xarray object from the pandas object.
<code>Series.to_hdf(path_or_buf, key, **kwargs)</code>	Write the contained data to an HDF5 file using HDFStore.
<code>Series.to_sql(name, con[, schema, ...])</code>	Write records stored in a DataFrame to a SQL database.
<code>Series.to_msgpack([path_or_buf, encoding])</code>	msgpack (serialize) object to input file path
<code>Series.to_json([path_or_buf, orient, ...])</code>	Convert the object to a JSON string.
<code>Series.to_sparse([kind, fill_value])</code>	Convert Series to SparseSeries
<code>Series.to_dense()</code>	Return dense representation of NDFrame (as opposed to sparse)
<code>Series.to_string([buf, na_rep, ...])</code>	Render a string representation of the Series
<code>Series.to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>Series.to_latex([buf, columns, col_space, ...])</code>	Render an object to a tabular environment table.

### 34.3.18 Sparse

<code>SparseSeries.to_coo([row_levels, ...])</code>	Create a <code>scipy.sparse.coo_matrix</code> from a <code>SparseSeries</code> with <code>MultiIndex</code> .
<code>SparseSeries.from_coo(A[, dense_index])</code>	Create a <code>SparseSeries</code> from a <code>scipy.sparse.coo_matrix</code> .

#### 34.3.18.1 pandas.SparseSeries.to\_coo

`SparseSeries.to_coo(row_levels=(0, ), column_levels=(1, ), sort_labels=False)`

Create a `scipy.sparse.coo_matrix` from a `SparseSeries` with `MultiIndex`.

Use `row_levels` and `column_levels` to determine the row and column coordinates respectively. `row_levels` and `column_levels` are the names (labels) or numbers of the levels. {`row_levels`, `column_levels`} must be a partition of the `MultiIndex` level names (or numbers).

**Parameters**

**row\_levels** [tuple/list]

**column\_levels** [tuple/list]

**sort\_labels** : bool, default False

Sort the row and column labels before forming the sparse matrix.

**Returns**

**y** [scipy.sparse.coo\_matrix]  
**rows** [list (row labels)]  
**columns** [list (column labels)]

**Examples**

```
>>> from numpy import nan
>>> s = Series([3.0, nan, 1.0, 3.0, nan, nan])
>>> s.index = MultiIndex.from_tuples([(1, 2, 'a', 0),
                                     (1, 2, 'a', 1),
                                     (1, 1, 'b', 0),
                                     (1, 1, 'b', 1),
                                     (2, 1, 'b', 0),
                                     (2, 1, 'b', 1)],
                                     names=['A', 'B', 'C', 'D'])

>>> ss = s.to_sparse()
>>> A, rows, columns = ss.to_coo(row_levels=['A', 'B'],
                                column_levels=['C', 'D'],
                                sort_labels=True)

>>> A
<3x4 sparse matrix of type '<class 'numpy.float64'>'
  with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  3.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
>>> rows
[(1, 1), (1, 2), (2, 1)]
>>> columns
[('a', 0), ('a', 1), ('b', 0), ('b', 1)]
```

### 34.3.18.2 pandas.SparseSeries.from\_coo

**classmethod** `SparseSeries.from_coo(A, dense_index=False)`

Create a SparseSeries from a scipy.sparse.coo\_matrix.

**Parameters**

**A** [scipy.sparse.coo\_matrix]  
**dense\_index** : bool, default False

If False (default), the SparseSeries index consists of only the coords of the non-null entries of the original coo\_matrix. If True, the SparseSeries index consists of the full sorted (row, col) coordinates of the coo\_matrix.

**Returns**

**s** [SparseSeries]

## Examples

```
>>> from scipy import sparse
>>> A = sparse.coo_matrix(([3.0, 1.0, 2.0], ([1, 0, 0], [0, 2, 3])),
                          shape=(3, 4))
>>> A
<3x4 sparse matrix of type '<class 'numpy.float64''>'
  with 3 stored elements in COOrdinate format>
>>> A.todense()
matrix([[ 0.,  0.,  1.,  2.],
        [ 3.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.]])
>>> ss = SparseSeries.from_coo(A)
>>> ss
0 2 1
3 2
1 0 3
dtype: float64
BlockIndex
Block locations: array([0], dtype=int32)
Block lengths: array([3], dtype=int32)
```

## 34.4 DataFrame

### 34.4.1 Constructor

<code>DataFrame([data, index, columns, dtype, copy])</code>	Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns).
---	--

#### 34.4.1.1 pandas.DataFrame

**class** pandas.DataFrame (*data=None, index=None, columns=None, dtype=None, copy=False*)

Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects. The primary pandas data structure.

**Parameters** **data** : numpy ndarray (structured or homogeneous), dict, or DataFrame

Dict can contain Series, arrays, constants, or list-like objects

Changed in version 0.23.0: If data is a dict, argument order is maintained for Python 3.6 and later.

**index** : Index or array-like

Index to use for resulting frame. Will default to RangeIndex if no indexing information part of input data and no index provided

**columns** : Index or array-like

Column labels to use for resulting frame. Will default to RangeIndex (0, 1, 2, ..., n) if no column labels are provided

**dtype** : dtype, default None

Data type to force. Only a single dtype is allowed. If None, infer

**copy** : boolean, default False

Copy data from inputs. Only affects DataFrame / 2d ndarray input

**See also:**

*DataFrame.from\_records* constructor from tuples, also record arrays

*DataFrame.from\_dict* from dicts of Series, arrays, or dicts

*DataFrame.from\_items* from sequence of (key, value) pairs

*pandas.read\_csv*, *pandas.read\_table*, *pandas.read\_clipboard*

## Examples

Constructing DataFrame from a dictionary.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df
   col1  col2
0     1     3
1     2     4
```

Notice that the inferred dtype is int64.

```
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

To enforce a single dtype:

```
>>> df = pd.DataFrame(data=d, dtype=np.int8)
>>> df.dtypes
col1    int8
col2    int8
dtype: object
```

Constructing DataFrame from numpy ndarray:

```
>>> df2 = pd.DataFrame(np.random.randint(low=0, high=10, size=(5, 5)),
...                     columns=['a', 'b', 'c', 'd', 'e'])
>>> df2
   a  b  c  d  e
0  2  8  8  3  4
1  4  2  9  0  9
2  1  0  7  8  0
3  5  1  7  1  3
4  6  0  2  4  2
```

## Attributes

<i>T</i>	Transpose index and columns.
<i>at</i>	Access a single value for a row/column label pair.
<i>axes</i>	Return a list representing the axes of the DataFrame.
<i>blocks</i>	(DEPRECATED) Internal property, property synonym for <i>as_blocks()</i>
<i>columns</i>	The column labels of the DataFrame.
<i>dtypes</i>	Return the dtypes in the DataFrame.
<i>empty</i>	Indicator whether DataFrame is empty.
<i>ftypes</i>	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
<i>iat</i>	Access a single value for a row/column pair by integer position.
<i>iloc</i>	Purely integer-location based indexing for selection by position.
<i>index</i>	The index (row labels) of the DataFrame.
<i>ix</i>	A primarily label-location based indexer, with integer position fallback.
<i>loc</i>	Access a group of rows and columns by label(s) or a boolean array.
<i>ndim</i>	Return an int representing the number of axes / array dimensions.
<i>shape</i>	Return a tuple representing the dimensionality of the DataFrame.
<i>size</i>	Return an int representing the number of elements in this object.
<i>style</i>	Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.
<i>values</i>	Return a Numpy representation of the DataFrame.

## pandas.DataFrame.T

`DataFrame.T`

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method *transpose()*.

**Parameters** *copy* : bool, default False

If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns** **DataFrame**

The transposed DataFrame.

**See also:**

`numpy.transpose` Permute the dimensions of a given array.

## Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

## Examples

### Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
   0  1
col1 1  2
col2 3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
0    int64
1    int64
dtype: object
```

### Non-square DataFrame with mixed dtypes

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
   name  score  employed  kids
0  Alice   9.5     False    0
1   Bob   8.0      True    0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
   0  1
name  Alice  Bob
score   9.5    8
employed False  True
kids      0    0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name          object
score        float64
employed      bool
kids          int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

## pandas.DataFrame.at

### DataFrame.at

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

#### Raises **KeyError**

When label does not exist in DataFrame

#### See also:

**DataFrame.iat** Access a single value for a row/column pair by integer position

**DataFrame.loc** Access a group of rows and columns by label(s)

**Series.at** Access a single value using a label

## Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
   A  B  C
4  0  2  3
5  0  4  1
6 10 20 30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

## pandas.DataFrame.axes

### DataFrame.axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

### Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

## pandas.DataFrame.blocks

### DataFrame.blocks

Internal property, property synonym for `as_blocks()`

Deprecated since version 0.21.0.

## pandas.DataFrame.columns

### DataFrame.columns

The column labels of the DataFrame.

## pandas.DataFrame.dtypes

### DataFrame.dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See [the User Guide](#) for more.

#### Returns pandas.Series

The data type of each column.

See also:

[`pandas.DataFrame.ftypes`](#) dtype and sparsity information.

### Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float          float64
int            int64
```

(continues on next page)



(continued from previous page)

```

datetime    datetime64[ns]
string      object
dtype: object

```

**pandas.DataFrame.empty****DataFrame.empty**

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

**Returns bool**

If DataFrame is empty, return True, if not return False.

**See also:***pandas.Series.dropna, pandas.DataFrame.dropna***Notes**

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

**Examples**

An example of an actual empty DataFrame. Notice the index is empty:

```

>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True

```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```

>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
   A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True

```

**pandas.DataFrame.ftypes****DataFrame.ftypes**

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See *the User Guide* for more.

**Returns** `pandas.Series`

The data type and indication of sparse/dense of each column.

**See also:**

`pandas.DataFrame.dtypes` Series with just dtype information.

`pandas.SparseDataFrame` Container for sparse tabular data.

**Notes**

Sparse data should have the same dtypes as its dense representation.

**Examples**

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).dtypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).dtypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

## **pandas.DataFrame.iat**

### **DataFrame.iat**

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

**Raises** `IndexError`

When integer position is out of bounds

**See also:**

`DataFrame.at` Access a single value for a row/column label pair

`DataFrame.loc` Access a group of rows and columns by label(s)

`DataFrame.iloc` Access a group of rows and columns by integer position(s)

## Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                     columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

## pandas.DataFrame.iloc

### DataFrame.iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at [Selection by Position](#)

## pandas.DataFrame.index

### DataFrame.index

The index (row labels) of the DataFrame.

## pandas.DataFrame.ix

### DataFrame.ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the .ix indexer is deprecated, in favor of the more strict .iloc and .loc indexers.

.ix[] supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

.ix is the most general indexer and will support any of the inputs in .loc and .iloc. .ix also supports floating point label schemes. .ix is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use .iloc or .loc.

See more at [Advanced Indexing](#).

## pandas.DataFrame.loc

### DataFrame.loc

Access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a':'f'.

**Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at [Selection by Label](#)

#### Raises KeyError:

when any items are not found

See also:

**DataFrame.at** Access a single value for a row/column label pair

**DataFrame.iloc** Access group of rows and columns by integer position(s)

**DataFrame.xs** Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

**Series.loc** Access group of values using labels

## Examples

### Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                     index=['cobra', 'viper', 'sidewinder'],
...                     columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using [ ] returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
```

	max_speed	shield
viper	4	5
sidewinder	7	8

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
```

	max_speed	shield
sidewinder	7	8

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
```

	max_speed	shield
sidewinder	7	8

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
```

	max_speed
sidewinder	7

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
      max_speed  shield
sidewinder      7      8
```

### Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
      max_speed  shield
cobra           1      2
viper           4     50
sidewinder      7     50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
      max_speed  shield
cobra         10     10
viper          4     50
sidewinder     7     50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
      max_speed  shield
cobra         30     10
viper         30     50
sidewinder    30     50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
      max_speed  shield
cobra         30     10
viper          0      0
sidewinder     0      0
```

### Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
      max_speed  shield
7           1      2
8           4      5
9           7      8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
      max_speed  shield
7             1       2
8             4       5
9             7       8
```

### Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
```

		max_speed	shield
cobra	mark i	12	2
	mark ii	0	4
sidewinder	mark i	10	20
	mark ii	1	4
viper	mark ii	7	1
	mark iii	16	36

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
      max_speed  shield
mark i         12       2
mark ii         0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield        4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using [ [] ] returns a DataFrame.

```
>>> df.loc[['cobra', 'mark ii']]
      max_speed  shield
cobra mark ii         0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
      max_speed  shield
cobra      mark i      12      2
           mark ii       0      4
sidewinder mark i      10     20
           mark ii       1      4
viper      mark ii       7      1
           mark iii      16     36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):(viper', 'mark ii')]
      max_speed  shield
cobra      mark i      12      2
           mark ii       0      4
sidewinder mark i      10     20
           mark ii       1      4
viper      mark ii       7      1
```

## pandas.DataFrame.ndim

`DataFrame.ndim`

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

**See also:**

`ndarray.ndim`

## Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

## pandas.DataFrame.shape

`DataFrame.shape`

Return a tuple representing the dimensionality of the DataFrame.

**See also:**

`ndarray.shape`



## Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                    'col3': [5, 6]})
>>> df.shape
(2, 3)
```

## pandas.DataFrame.size

### DataFrame.size

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

#### See also:

`ndarray.size`

## Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

## pandas.DataFrame.style

### DataFrame.style

Property returning a Styler object containing methods for building a styled HTML representation for the DataFrame.

#### See also:

`pandas.io.formats.style.Styler`

## pandas.DataFrame.values

### DataFrame.values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

#### Returns `numpy.ndarray`

The values of the DataFrame.

See also:

`pandas.DataFrame.index` Retrieve the index labels

`pandas.DataFrame.columns` Retrieving the column names

## Notes

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

## Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age': [ 3, 29],
...                     'height': [94, 170],
...                     'weight': [31, 115]})
>>> df
   age  height  weight
0    3     94     31
1   29    170    115
>>> df.dtypes
age      int64
height   int64
weight   int64
dtype: object
>>> df.values
array([[ 3, 94, 31],
       [29, 170, 115]], dtype=int64)
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot', 24.0, 'second'),
...                     ('lion', 80.5, 1),
...                     ('monkey', np.nan, None)],
...                     columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name      object
max_speed  float64
rank      object
dtype: object
>>> df2.values
array([('parrot', 24.0, 'second'],
      ['lion', 80.5, 1],
      ['monkey', nan, None]], dtype=object)
```

is_copy	
---------	--

## Methods

<code>abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>add(other[, axis, level, fill_value])</code>	Addition of dataframe and other, element-wise (binary operator <i>add</i> ).
<code>add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>agg(func[, axis])</code>	Aggregate using one or more operations over the specified axis.
<code>aggregate(func[, axis])</code>	Aggregate using one or more operations over the specified axis.
<code>align(other[, join, axis, level, copy, ...])</code>	Align two objects on their axes with the specified join method for each axis Index
<code>all([axis, bool_only, skipna, level])</code>	Return whether all elements are True, potentially over an axis.
<code>any([axis, bool_only, skipna, level])</code>	Return whether any element is True over requested axis.
<code>append(other[, ignore_index, ...])</code>	Append rows of <i>other</i> to the end of this frame, returning a new object.
<code>apply(func[, axis, broadcast, raw, reduce, ...])</code>	Apply a function along an axis of the DataFrame.
<code>applymap(func)</code>	Apply a function to a Dataframe elementwise.
<code>as_blocks([copy])</code>	(DEPRECATED) Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
<code>as_matrix([columns])</code>	(DEPRECATED) Convert the frame to its Numpy-array representation.
<code>asfreq(freq[, method, how, normalize, ...])</code>	Convert TimeSeries to specified frequency.
<code>asof(where[, subset])</code>	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
<code>assign(**kwargs)</code>	Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones.
<code>astype(dtype[, copy, errors])</code>	Cast a pandas object to a specified dtype <i>dtype</i> .
<code>at_time(time[, asof])</code>	Select values at particular time of day (e.g.
<code>between_time(start_time, end_time[, ...])</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill([axis, inplace, limit, downcast])</code>	Synonym for <code>DataFrame.fillna(method='bfill')</code>
<code>bool()</code>	Return the bool of a single element PandasObject.
<code>boxplot([column, by, ax, fontsize, rot, ...])</code>	Make a box plot from DataFrame columns.
<code>clip([lower, upper, axis, inplace])</code>	Trim values at input threshold(s).
<code>clip_lower(threshold[, axis, inplace])</code>	Return copy of the input with values below a threshold truncated.
<code>clip_upper(threshold[, axis, inplace])</code>	Return copy of input with values above given value(s) truncated.
<code>combine(other, func[, fill_value, overwrite])</code>	Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

Continued on next page

Table 61 – continued from previous page

<code>combine_first(other)</code>	Combine two DataFrame objects and default to non-null values in frame calling the method.
<code>compound([axis, skipna, level])</code>	Return the compound percentage of the values for the requested axis
<code>consolidate([inplace])</code>	(DEPRECATED) Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray).
<code>convert_objects([convert_dates, ...])</code>	(DEPRECATED) Attempt to infer better dtype for object columns.
<code>copy([deep])</code>	Make a copy of this object’s indices and data.
<code>corr([method, min_periods])</code>	Compute pairwise correlation of columns, excluding NA/null values
<code>corrwith(other[, axis, drop])</code>	Compute pairwise correlation between rows or columns of two DataFrame objects.
<code>count([axis, level, numeric_only])</code>	Count non-NA cells for each column or row.
<code>cov([min_periods])</code>	Compute pairwise covariance of columns, excluding NA/null values.
<code>cummax([axis, skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin([axis, skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod([axis, skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>cumsum([axis, skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>describe([percentiles, include, exclude])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.
<code>diff([periods, axis])</code>	First discrete difference of element.
<code>div(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>divide(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>dot(other)</code>	Matrix multiplication with DataFrame or Series objects.
<code>drop([labels, axis, index, columns, level, ...])</code>	Drop specified labels from rows or columns.
<code>drop_duplicates([subset, keep, inplace])</code>	Return DataFrame with duplicate rows removed, optionally only considering certain columns
<code>dropna([axis, how, thresh, subset, inplace])</code>	Remove missing values.
<code>duplicated([subset, keep])</code>	Return boolean Series denoting duplicate rows, optionally only considering certain columns
<code>eq(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>eq</i>
<code>equals(other)</code>	Determines if two NDFrame objects contain the same elements.
<code>eval(expr[, inplace])</code>	Evaluate a string describing operations on DataFrame columns.
<code>ewm([com, span, halflife, alpha, ...])</code>	Provides exponential weighted functions
<code>expanding([min_periods, center, axis])</code>	Provides expanding transformations.
<code>ffill([axis, inplace, limit, downcast])</code>	Synonym for <code>DataFrame.fillna(method='ffill')</code>
<code>fillna([value, method, axis, inplace, ...])</code>	Fill NA/NaN values using the specified method

Continued on next page

Table 61 – continued from previous page

<code>filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>first_valid_index()</code>	Return index for first non-NA/null value.
<code>floordiv(other[, axis, level, fill_value])</code>	Integer division of dataframe and other, element-wise (binary operator <i>floordiv</i> ).
<code>from_csv(path[, header, sep, index_col, ...])</code>	(DEPRECATED) Read CSV file.
<code>from_dict(data[, orient, dtype, columns])</code>	Construct DataFrame from dict of array-like or dicts.
<code>from_items(items[, columns, orient])</code>	(DEPRECATED) Construct a dataframe from a list of tuples
<code>from_records(data[, index, exclude, ...])</code>	Convert structured or record ndarray to DataFrame
<code>ge(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>ge</i>
<code>get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>get_dtype_counts()</code>	Return counts of unique dtypes in this object.
<code>get_ftype_counts()</code>	(DEPRECATED) Return counts of unique ftypes in this object.
<code>get_value(index, col[, takeable])</code>	(DEPRECATED) Quickly retrieve single value at passed column and index
<code>get_values()</code>	Return an ndarray after converting sparse values to dense.
<code>groupby([by, axis, level, as_index, sort, ...])</code>	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
<code>gt(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>gt</i>
<code>head([n])</code>	Return the first <i>n</i> rows.
<code>hist([column, by, grid, xlabelsize, xrot, ...])</code>	Make a histogram of the DataFrame's.
<code>idxmax([axis, skipna])</code>	Return index of first occurrence of maximum over requested axis.
<code>idxmin([axis, skipna])</code>	Return index of first occurrence of minimum over requested axis.
<code>infer_objects()</code>	Attempt to infer better dtypes for object columns.
<code>info([verbose, buf, max_cols, memory_usage, ...])</code>	Print a concise summary of a DataFrame.
<code>insert(loc, column, value[, allow_duplicates])</code>	Insert column into DataFrame at specified location.
<code>interpolate([method, axis, limit, inplace, ...])</code>	Interpolate values according to different methods.
<code>isin(values)</code>	Return boolean DataFrame showing whether each element in the DataFrame is contained in values.
<code>isna()</code>	Detect missing values.
<code>isnull()</code>	Detect missing values.
<code>items()</code>	Iterator over (column name, Series) pairs.
<code>iteritems()</code>	Iterator over (column name, Series) pairs.
<code>iterrows()</code>	Iterate over DataFrame rows as (index, Series) pairs.
<code>itertuples([index, name])</code>	Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.
<code>join(other[, on, how, lsuffix, rsuffix, sort])</code>	Join columns with other DataFrame either on index or on a key column.
<code>keys()</code>	Get the 'info axis' (see Indexing for more)

Continued on next page

Table 61 – continued from previous page

<code>kurt([axis, skipna, level, numeric_only])</code>	Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).
<code>kurtosis([axis, skipna, level, numeric_only])</code>	Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).
<code>last(offset)</code>	Convenience method for subsetting final periods of time series data based on a date offset.
<code>last_valid_index()</code>	Return index for last non-NA/null value.
<code>le(other[, axis, level])</code>	Wrapper for flexible comparison methods <code>le</code>
<code>lookup(row_labels, col_labels)</code>	Label-based “fancy indexing” function for DataFrame.
<code>lt(other[, axis, level])</code>	Wrapper for flexible comparison methods <code>lt</code>
<code>mad([axis, skipna, level])</code>	Return the mean absolute deviation of the values for the requested axis
<code>mask(cond[, other, inplace, axis, level, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>max([axis, skipna, level, numeric_only])</code>	This method returns the maximum of the values in the object.
<code>mean([axis, skipna, level, numeric_only])</code>	Return the mean of the values for the requested axis
<code>median([axis, skipna, level, numeric_only])</code>	Return the median of the values for the requested axis
<code>melt([id_vars, value_vars, var_name, ...])</code>	“Unpivots” a DataFrame from wide format to long format, optionally leaving identifier variables set.
<code>memory_usage([index, deep])</code>	Return the memory usage of each column in bytes.
<code>merge(right[, how, on, left_on, right_on, ...])</code>	Merge DataFrame objects by performing a database-style join operation by columns or indexes.
<code>min([axis, skipna, level, numeric_only])</code>	This method returns the minimum of the values in the object.
<code>mod(other[, axis, level, fill_value])</code>	Modulo of dataframe and other, element-wise (binary operator <i>mod</i> ).
<code>mode([axis, numeric_only])</code>	Gets the mode(s) of each element along the axis selected.
<code>mul(other[, axis, level, fill_value])</code>	Multiplication of dataframe and other, element-wise (binary operator <i>mul</i> ).
<code>multiply(other[, axis, level, fill_value])</code>	Multiplication of dataframe and other, element-wise (binary operator <i>mul</i> ).
<code>ne(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>ne</i>
<code>nlargest(n, columns[, keep])</code>	Return the first <i>n</i> rows ordered by <i>columns</i> in descending order.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nsmallest(n, columns[, keep])</code>	Get the rows of a DataFrame sorted by the <i>n</i> smallest values of <i>columns</i> .
<code>nunique([axis, dropna])</code>	Return Series with number of distinct observations over requested axis.
<code>pct_change([periods, fill_method, limit, freq])</code>	Percentage change between the current and a prior element.
<code>pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code>

Continued on next page

Table 61 – continued from previous page

<code>pivot([index, columns, values])</code>	Return reshaped DataFrame organized by given index / column values.
<code>pivot_table([values, index, columns, ...])</code>	Create a spreadsheet-style pivot table as a DataFrame.
<code>plot</code>	alias of <code>pandas.plotting._core.FramePlotMethods</code>
<code>pop(item)</code>	Return item and drop from frame.
<code>pow(other[, axis, level, fill_value])</code>	Exponential power of dataframe and other, element-wise (binary operator <code>pow</code> ).
<code>prod([axis, skipna, level, numeric_only, ...])</code>	Return the product of the values for the requested axis
<code>product([axis, skipna, level, numeric_only, ...])</code>	Return the product of the values for the requested axis
<code>quantile([q, axis, numeric_only, interpolation])</code>	Return values at the given quantile over requested axis, a la <code>numpy.percentile</code> .
<code>query(expr[, inplace])</code>	Query the columns of a frame with a boolean expression.
<code>radd(other[, axis, level, fill_value])</code>	Addition of dataframe and other, element-wise (binary operator <code>radd</code> ).
<code>rank([axis, method, numeric_only, ...])</code>	Compute numerical data ranks (1 through n) along axis.
<code>rdiv(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <code>rtruediv</code> ).
<code>reindex([labels, index, columns, axis, ...])</code>	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_axis(labels[, axis, method, level, ...])</code>	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_like(other[, method, copy, limit, ...])</code>	Return an object with matching indices to myself.
<code>rename([mapper, index, columns, axis, copy, ...])</code>	Alter axes labels.
<code>rename_axis(mapper[, axis, copy, inplace])</code>	Alter the name of the index or columns.
<code>reorder_levels(order[, axis])</code>	Rearrange index levels using input order.
<code>replace([to_replace, value, inplace, limit, ...])</code>	Replace values given in <code>to_replace</code> with <code>value</code> .
<code>resample(rule[, how, axis, fill_method, ...])</code>	Convenience method for frequency conversion and resampling of time series.
<code>reset_index([level, drop, inplace, ...])</code>	For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc.
<code>rfloordiv(other[, axis, level, fill_value])</code>	Integer division of dataframe and other, element-wise (binary operator <code>rfloordiv</code> ).
<code>rmod(other[, axis, level, fill_value])</code>	Modulo of dataframe and other, element-wise (binary operator <code>rmod</code> ).
<code>rmul(other[, axis, level, fill_value])</code>	Multiplication of dataframe and other, element-wise (binary operator <code>rmul</code> ).
<code>rolling(window[, min_periods, center, ...])</code>	Provides rolling window calculations.
<code>round([decimals])</code>	Round a DataFrame to a variable number of decimal places.
<code>rpow(other[, axis, level, fill_value])</code>	Exponential power of dataframe and other, element-wise (binary operator <code>rpow</code> ).

Continued on next page

Table 61 – continued from previous page

<code>rsub</code> (other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator <i>rsub</i> ).
<code>rtruediv</code> (other[, axis, level, fill_value])	Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).
<code>sample</code> ([n, frac, replace, weights, ...])	Return a random sample of items from an axis of object.
<code>select</code> (crit[, axis])	(DEPRECATED) Return data corresponding to axis labels matching criteria
<code>select_dtypes</code> ([include, exclude])	Return a subset of the DataFrame's columns based on the column dtypes.
<code>sem</code> ([axis, skipna, level, ddof, numeric_only])	Return unbiased standard error of the mean over requested axis.
<code>set_axis</code> (labels[, axis, inplace])	Assign desired index to given axis.
<code>set_index</code> (keys[, drop, append, inplace, ...])	Set the DataFrame index (row labels) using one or more existing columns.
<code>set_value</code> (index, col, value[, takeable])	(DEPRECATED) Put single value at passed column and index
<code>shift</code> ([periods, freq, axis])	Shift index by desired number of periods with an optional time freq
<code>skew</code> ([axis, skipna, level, numeric_only])	Return unbiased skew over requested axis Normalized by N-1
<code>slice_shift</code> ([periods, axis])	Equivalent to <i>shift</i> without copying data.
<code>sort_index</code> ([axis, level, ascending, ...])	Sort object by labels (along an axis)
<code>sort_values</code> (by[, axis, ascending, inplace, ...])	Sort by the values along either axis
<code>sortlevel</code> ([level, axis, ascending, inplace, ...])	(DEPRECATED) Sort multilevel index by chosen axis and primary level.
<code>squeeze</code> ([axis])	Squeeze length 1 dimensions.
<code>stack</code> ([level, dropna])	Stack the prescribed level(s) from columns to index.
<code>std</code> ([axis, skipna, level, ddof, numeric_only])	Return sample standard deviation over requested axis.
<code>sub</code> (other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator <i>sub</i> ).
<code>subtract</code> (other[, axis, level, fill_value])	Subtraction of dataframe and other, element-wise (binary operator <i>sub</i> ).
<code>sum</code> ([axis, skipna, level, numeric_only, ...])	Return the sum of the values for the requested axis
<code>swapaxes</code> (axis1, axis2[, copy])	Interchange axes and swap values axes appropriately
<code>swaplevel</code> ([i, j, axis])	Swap levels i and j in a MultiIndex on a particular axis
<code>tail</code> ([n])	Return the last <i>n</i> rows.
<code>take</code> (indices[, axis, convert, is_copy])	Return the elements in the given <i>positional</i> indices along an axis.
<code>to_clipboard</code> ([excel, sep])	Copy object to the system clipboard.
<code>to_csv</code> ([path_or_buf, sep, na_rep, ...])	Write DataFrame to a comma-separated values (csv) file
<code>to_dense</code> ()	Return dense representation of NDFrame (as opposed to sparse)
<code>to_dict</code> ([orient, into])	Convert the DataFrame to a dictionary.
<code>to_excel</code> (excel_writer[, sheet_name, na_rep, ...])	Write DataFrame to an excel sheet
<code>to_feather</code> (fname)	write out the binary feather-format for DataFrames
<code>to_gbq</code> (destination_table, project_id[, ...])	Write a DataFrame to a Google BigQuery table.

Continued on next page



Table 61 – continued from previous page

<code>to_hdf(path_or_buf, key, **kwargs)</code>	Write the contained data to an HDF5 file using HDF-Store.
<code>to_html([buf, columns, col_space, header, ...])</code>	Render a DataFrame as an HTML table.
<code>to_json([path_or_buf, orient, date_format, ...])</code>	Convert the object to a JSON string.
<code>to_latex([buf, columns, col_space, header, ...])</code>	Render an object to a tabular environment table.
<code>to_msgpack([path_or_buf, encoding])</code>	msgpack (serialize) object to input file path
<code>to_panel()</code>	(DEPRECATED) Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.
<code>to_parquet(fname[, engine, compression])</code>	Write a DataFrame to the binary parquet format.
<code>to_period([freq, axis, copy])</code>	Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)
<code>to_pickle(path[, compression, protocol])</code>	Pickle (serialize) object to file.
<code>to_records([index, convert_datetime64])</code>	Convert DataFrame to a NumPy record array.
<code>to_sparse([fill_value, kind])</code>	Convert to SparseDataFrame
<code>to_sql(name, con[, schema, if_exists, ...])</code>	Write records stored in a DataFrame to a SQL database.
<code>to_stata(fname[, convert_dates, ...])</code>	Export Stata binary dta files.
<code>to_string([buf, columns, col_space, header, ...])</code>	Render a DataFrame to a console-friendly tabular output.
<code>to_timestamp([freq, how, axis, copy])</code>	Cast to DatetimeIndex of timestamps, at <i>beginning</i> of period
<code>to_xarray()</code>	Return an xarray object from the pandas object.
<code>transform(func, *args, **kwargs)</code>	Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
<code>transpose(*args, **kwargs)</code>	Transpose index and columns.
<code>truediv(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.
<code>tshift([periods, freq, axis])</code>	Shift the time index, using the index's frequency if available.
<code>tz_convert(tz[, axis, level, copy])</code>	Convert tz-aware axis to target time zone.
<code>tz_localize(tz[, axis, level, copy, ambiguous])</code>	Localize tz-naive TimeSeries to target time zone.
<code>unstack([level, fill_value])</code>	Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.
<code>update(other[, join, overwrite, ...])</code>	Modify in place using non-NA values from another DataFrame.
<code>var([axis, skipna, level, ddof, numeric_only])</code>	Return unbiased variance over requested axis.
<code>where(cond[, other, inplace, axis, level, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .
<code>xs(key[, axis, level, drop_level])</code>	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

**pandas.DataFrame.abs**DataFrame.**abs** ()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

**Returns** `abs`

Series/DataFrame containing the absolute value of each element.

**See also:**

`numpy.absolute` calculate the absolute value element-wise.

**Notes**

For complex inputs,  $1.2 + 1j$ , the absolute value is  $\sqrt{a^2 + b^2}$ .

**Examples**

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0    1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using `argsort` (from [StackOverflow](#)).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
   a  b  c
0  4 10 100
1  5 20  50
2  6 30 -30
3  7 40 -50
>>> df.loc[(df.c - 43).abs().argsort()]
   a  b  c
1  5 20  50
```

(continues on next page)

(continued from previous page)

0	4	10	100
2	6	30	-30
3	7	40	-50

**pandas.DataFrame.add**`DataFrame.add(other, axis='columns', level=None, fill_value=None)`Addition of dataframe and other, element-wise (binary operator *add*).Equivalent to `dataframe + other`, but with support to substitute a `fill_value` for missing data in one of the inputs.**Parameters****other** [Series, DataFrame, or constant]**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

**Returns****result** [DataFrame]**See also:**`DataFrame.radd`**Notes**

Mismatched indices will be unioned together

**Examples**

```

>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b

```

(continues on next page)

(continued from previous page)

```

      one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
      one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0

```

**pandas.DataFrame.add\_prefix****DataFrame.add\_prefix** (*prefix*)Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters** *prefix* : str

The string to add before each label.

**Returns** **Series or DataFrame**

New Series or DataFrame with updated labels.

**See also:****Series.add\_suffix** Suffix row labels with string *suffix*.**DataFrame.add\_suffix** Suffix column labels with string *suffix*.**Examples**

```

>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64

```

```

>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64

```

```

>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B

```

(continues on next page)

(continued from previous page)

```
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
      col_A  col_B
0         1     3
1         2     4
2         3     5
3         4     6
```

**pandas.DataFrame.add\_suffix****DataFrame.add\_suffix** (*suffix*)Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

**Parameters** *suffix* : str

The string to add after each label.

**Returns** Series or DataFrame

New Series or DataFrame with updated labels.

**See also:****Series.add\_prefix** Prefix row labels with string *prefix*.**DataFrame.add\_prefix** Prefix column labels with string *prefix*.**Examples**

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
```

(continues on next page)

(continued from previous page)

```
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
      A_col  B_col
0         1     3
1         2     4
2         3     5
3         4     6
```

## pandas.DataFrame.agg

`DataFrame.agg(func, axis=0, *args, **kwargs)`

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

**Parameters** **func** : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**axis** : {0 or 'index', 1 or 'columns'}, default 0

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.

**Returns**

**aggregated** [DataFrame]

**See also:**

**DataFrame.apply** Perform any type of operations.

**DataFrame.transform** Perform transformation type operations.

**pandas.core.groupby.GroupBy** Perform operations over groups.

**pandas.core.resample.Resampler** Perform operations over resampled bins.

**pandas.core.window.Rolling** Perform operations over rolling window.

**pandas.core.window.Expanding** Perform operations over expanding window.

**pandas.core.window.EWM** Perform operation over exponential weighted window.

## Notes

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

*agg* is an alias for *aggregate*. Use the alias.

## Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
      A      B      C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
      A      B
max  NaN  8.0
min   1.0  2.0
sum  12.0 NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0      2.0
1      5.0
2      8.0
3      NaN
dtype: float64
```

## pandas.DataFrame.aggregate

`DataFrame.aggregate(func, axis=0, *args, **kwargs)`

Aggregate using one or more operations over the specified axis.

New in version 0.20.0.

**Parameters** **func** : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**axis** : {0 or 'index', 1 or 'columns'}, default 0

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.

**Returns**

**aggregated** [DataFrame]

**See also:**

**DataFrame.apply** Perform any type of operations.

**DataFrame.transform** Perform transformation type operations.

**pandas.core.groupby.GroupBy** Perform operations over groups.

**pandas.core.resample.Resampler** Perform operations over resampled bins.

**pandas.core.window.Rolling** Perform operations over rolling window.

**pandas.core.window.Expanding** Perform operations over expanding window.

**pandas.core.window.EWM** Perform operation over exponential weighted window.

## Notes

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

The aggregation operations are always performed over an axis, either the index (default) or the column axis. This behavior is different from *numpy* aggregation functions (*mean*, *median*, *prod*, *sum*, *std*, *var*), where the default is to compute the aggregation of the flattened array, e.g., `numpy.mean(arr_2d)` as opposed to `numpy.mean(arr_2d, axis=0)`.

*agg* is an alias for *aggregate*. Use the alias.



## Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                     [4, 5, 6],
...                     [7, 8, 9],
...                     [np.nan, np.nan, np.nan]],
...                    columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
      A      B      C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
      A      B
max  NaN   8.0
min   1.0   2.0
sum  12.0  NaN
```

Aggregate over the columns.

```
>>> df.agg("mean", axis="columns")
0      2.0
1      5.0
2      8.0
3      NaN
dtype: float64
```

## pandas.DataFrame.align

`DataFrame.align(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0, broadcast_axis=None)`

Align two objects on their axes with the specified join method for each axis Index

### Parameters

**other** [DataFrame or Series]

**join** [{‘outer’, ‘inner’, ‘left’, ‘right’}, default ‘outer’]

**axis** : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

**level** : int or level name, default None

Broadcast across a level, matching Index values on the passed MultiIndex level

**copy** : boolean, default True

Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any “compatible” value

**method** [str, default None]

**limit** [int, default None]

**fill\_axis** : {0 or ‘index’, 1 or ‘columns’}, default 0

Filling axis, method and limit

**broadcast\_axis** : {0 or ‘index’, 1 or ‘columns’}, default None

Broadcast values along this axis, if aligning two objects of different dimensions

**Returns** (**left**, **right**) : (DataFrame, type of other)

Aligned objects

## **pandas.DataFrame.all**

`DataFrame.all` (*axis=0*, *bool\_only=None*, *skipna=True*, *level=None*, *\*\*kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

**Parameters** **axis** : {0 or ‘index’, 1 or ‘columns’, None}, default 0

Indicate which axis or axes should be reduced.

- 0 / ‘index’ : reduce the index, return a Series whose index is the original column labels.
- 1 / ‘columns’ : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**\*\*kwargs** : any, default None

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### **Returns**

**all** [Series or DataFrame (if level specified)]

**See also:**

[`pandas.Series.all`](#) Return True if all elements are True

**pandas.DataFrame.any** Return True if one (or more) elements are True

## Examples

### Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

### DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1  col2
0  True  True
1  True False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1    True
col2   False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0    True
1   False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

## pandas.DataFrame.any

`DataFrame.any` (*axis=0*, *bool\_only=None*, *skipna=True*, *level=None*, *\*\*kwargs*)

Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

**Parameters** `axis`: {0 or 'index', 1 or 'columns', None}, default 0

Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.

- None : reduce all axes, return a scalar.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series.

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**\*\*kwargs** : any, default None

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**any** [Series or DataFrame (if level specified)]

**See also:**

[`pandas.DataFrame.all`](#) Return whether all elements are True.

## Examples

### Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

### DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A      True
B      True
C     False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
   A  B
0  True  1
1 False  2
```

```
>>> df.any(axis='columns')
0      True
1      True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
   A  B
0  True  1
1 False  0
```

```
>>> df.any(axis='columns')
0      True
1     False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

`any` for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

## pandas.DataFrame.append

`DataFrame.append(other, ignore_index=False, verify_integrity=False, sort=None)`

Append rows of *other* to the end of this frame, returning a new object. Columns not in this frame are added as new columns.

**Parameters** **other** : DataFrame or Series/dict-like object, or list of these

The data to append.

**ignore\_index** : boolean, default False

If True, do not use the index labels.

**verify\_integrity** : boolean, default False

If True, raise `ValueError` on creating index with duplicates.

**sort** : boolean, default None

Sort columns if the columns of *self* and *other* are not aligned. The default sorting is deprecated and will change to not-sorting in a future version of pandas. Explicitly pass `sort=True` to silence the warning and sort. Explicitly pass `sort=False` to silence the warning and not sort.

New in version 0.23.0.

**Returns**

**appended** [DataFrame]

See also:

**pandas.concat** General function to concatenate DataFrame, Series or Panel objects

## Notes

If a list of dict/series is passed and the keys are all contained in the DataFrame's index, the order of the columns in the resulting DataFrame will be unchanged.

Iteratively appending rows to a DataFrame can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original DataFrame all at once.

## Examples

```
>>> df = pd.DataFrame([[1, 2], [3, 4]], columns=list('AB'))
>>> df
   A  B
0  1  2
1  3  4
>>> df2 = pd.DataFrame([[5, 6], [7, 8]], columns=list('AB'))
>>> df.append(df2)
   A  B
0  1  2
1  3  4
0  5  6
1  7  8
```

With `ignore_index` set to `True`:

```
>>> df.append(df2, ignore_index=True)
   A  B
0  1  2
1  3  4
2  5  6
3  7  8
```

The following, while not recommended methods for generating DataFrames, show two ways to generate a DataFrame from multiple data sources.

Less efficient:

```
>>> df = pd.DataFrame(columns=['A'])
>>> for i in range(5):
...     df = df.append({'A': i}, ignore_index=True)
>>> df
   A
0  0
1  1
2  2
3  3
4  4
```

More efficient:

```
>>> pd.concat([pd.DataFrame([i], columns=['A']) for i in range(5)],
...           ignore_index=True)
```

(continues on next page)

(continued from previous page)

	A
0	0
1	1
2	2
3	3
4	4

**pandas.DataFrame.apply**

`DataFrame.apply(func, axis=0, broadcast=None, raw=False, reduce=None, result_type=None, args=(), **kwargs)`

Apply a function along an axis of the DataFrame.

Objects passed to the function are Series objects whose index is either the DataFrame's index (`axis=0`) or the DataFrame's columns (`axis=1`). By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the `result_type` argument.

**Parameters** `func` : function

Function to apply to each column or row.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Axis along which the function is applied:

- 0 or 'index': apply function to each column.
- 1 or 'columns': apply function to each row.

**broadcast** : bool, optional

Only relevant for aggregation functions:

- `False` or `None` : returns a Series whose length is the length of the index or the number of columns (based on the `axis` parameter)
- `True` : results will be broadcast to the original shape of the frame, the original index and columns will be retained.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='broadcast'`.

**raw** : bool, default False

- `False` : passes each row or column as a Series to the function.
- `True` : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

**reduce** : bool or None, default None

Try to apply reduction procedures. If the DataFrame is empty, `apply` will use `reduce` to determine whether the result should be a Series or a DataFrame. If `reduce=None` (the default), `apply`'s return value will be guessed by calling `func` on an empty Series (note: while guessing, exceptions raised by `func` will be ignored). If `reduce=True` a Series will always be returned, and if `reduce=False` a DataFrame will always be returned.

Deprecated since version 0.23.0: This argument will be removed in a future version, replaced by `result_type='reduce'`.

**result\_type** : {'expand', 'reduce', 'broadcast', None}, default None

These only act when `axis=1` (columns):

- 'expand' : list-like results will be turned into columns.
- 'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.
- 'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.

The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.

New in version 0.23.0.

**args** : tuple

Positional arguments to pass to *func* in addition to the array/series.

**\*\*kwds**

Additional keyword arguments to pass as keywords arguments to *func*.

### Returns

**applied** [Series or DataFrame]

**See also:**

[\*DataFrame.applymap\*](#) For elementwise operations

[\*DataFrame.agg\*](#) only perform aggregating type operations

[\*DataFrame.transform\*](#) only perform transforming type operations

### Notes

In the current implementation `apply` calls *func* twice on the first column/row to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if *func* has side-effects, as they will take effect twice for the first column/row.

### Examples

```
>>> df = pd.DataFrame([[4, 9],] * 3, columns=['A', 'B'])
>>> df
   A  B
0  4  9
1  4  9
2  4  9
```

Using a numpy universal function (in this case the same as `np.sqrt(df)`):

```
>>> df.apply(np.sqrt)
   A    B
0  2.0  3.0
1  2.0  3.0
2  2.0  3.0
```



Using a reducing function on either axis

```
>>> df.apply(np.sum, axis=0)
A    12
B    27
dtype: int64
```

```
>>> df.apply(np.sum, axis=1)
0    13
1    13
2    13
dtype: int64
```

Returning a list-like will result in a Series

```
>>> df.apply(lambda x: [1, 2], axis=1)
0    [1, 2]
1    [1, 2]
2    [1, 2]
dtype: object
```

Passing `result_type='expand'` will expand list-like results to columns of a DataFrame

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='expand')
0  1
0  1  2
1  1  2
2  1  2
```

Returning a Series inside the function is similar to passing `result_type='expand'`. The resulting column names will be the Series index.

```
>>> df.apply(lambda x: pd.Series([1, 2], index=['foo', 'bar']), axis=1)
   foo  bar
0     1    2
1     1    2
2     1    2
```

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
>>> df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
   A  B
0  1  2
1  1  2
2  1  2
```

## pandas.DataFrame.applymap

`DataFrame.applymap(func)`

Apply a function to a DataFrame elementwise.

This method applies a function that accepts and returns a scalar to every element of a DataFrame.

**Parameters** `func` : callable

Python function, returns a single value from a single value.

**Returns DataFrame**

Transformed DataFrame.

**See also:**

**`DataFrame.apply`** Apply a function along input axis of DataFrame

**Examples**

```
>>> df = pd.DataFrame([[1, 2.12], [3.356, 4.567]])
>>> df
   0      1
0  1.000  2.120
1  3.356  4.567
```

```
>>> df.applymap(lambda x: len(str(x)))
   0  1
0  3  4
1  5  5
```

Note that a vectorized version of *func* often exists, which will be much faster. You could square each number elementwise.

```
>>> df.applymap(lambda x: x**2)
   0      1
0  1.000000  4.494400
1 11.262736 20.857489
```

But it's better to avoid `applymap` in that case.

```
>>> df ** 2
   0      1
0  1.000000  4.494400
1 11.262736 20.857489
```

**pandas.DataFrame.as\_blocks**

`DataFrame.as_blocks` (*copy=True*)

Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

**NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in `as_matrix`)**

**Parameters**

**copy** [boolean, default True]

**Returns**

**values** [a dict of dtype -> Constructor Types]

**pandas.DataFrame.as\_matrix**`DataFrame.as_matrix (columns=None)`

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.**Parameters** `columns`: list, optional, default:None

If None, return all columns, otherwise, returns specified columns.

**Returns** `values` : ndarray

If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

**See also:**`pandas.DataFrame.values`**Notes**

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By `numpy.find_common_type` convention, mixing int64 and uint64 will result in a float64 dtype.This method is provided for backwards compatibility. Generally, it is recommended to use `‘.values’`.**pandas.DataFrame.asfreq**`DataFrame.asfreq (freq, method=None, how=None, normalize=False, fill_value=None)`

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.**Parameters****freq** [DateOffset object, or string]**method** : {‘backfill’/‘bfill’, ‘pad’/‘ffill’}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- ‘pad’ / ‘ffill’: propagate last valid observation forward to next valid
- ‘backfill’ / ‘bfill’: use NEXT valid observation to fill

**how** : {‘start’, ‘end’}, default end

For PeriodIndex only, see PeriodIndex.asfreq

**normalize** : bool, default False

Whether to reset output index to midnight

**fill\_value:** scalar, optional

Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

### Returns

**converted** [type of caller]

**See also:**

[\*reindex\*](#)

### Notes

To learn more about the frequency strings, please see [this link](#).

### Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
```

	s
2000-01-01 00:00:00	0.0
2000-01-01 00:01:00	NaN
2000-01-01 00:02:00	2.0
2000-01-01 00:03:00	3.0

Upsample the series into 30 second bins.

```
>>> df.asfreq(freq='30S')
s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    NaN
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    NaN
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    NaN
2000-01-01 00:03:00    3.0
```

Upsample again, providing a fill value.

```
>>> df.asfreq(freq='30S', fill_value=9.0)
s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30    9.0
2000-01-01 00:01:00    NaN
2000-01-01 00:01:30    9.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    9.0
2000-01-01 00:03:00    3.0
```

Upsample again, providing a method.

```
>>> df.asfreq(freq='30S', method='bfill')
s
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00   NaN
2000-01-01 00:01:30    2.0
2000-01-01 00:02:00    2.0
2000-01-01 00:02:30    3.0
2000-01-01 00:03:00    3.0
```

## pandas.DataFrame.asof

`DataFrame.asof` (*where*, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

### Parameters

**where** [date or array of dates]

**subset** : string or list of strings, default None

if not None use these columns for NaN propagation

### Returns where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

**where is Index: same shape object as input**

**See also:**

[`merge\_asof`](#)

### Notes

Dates are assumed to be sorted Raises if this is not the case

## pandas.DataFrame.assign

`DataFrame.assign` (*\*\*kwargs*)

Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones. Existing columns that are re-assigned will be overwritten.

**Parameters** *kwargs* : keyword, value pairs

keywords are the column names. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

**Returns** `df` : DataFrame

A new DataFrame with the new columns in addition to all the existing columns.

**Notes**

Assigning multiple columns within the same `assign` is possible. For Python 3.6 and above, later items in `**kwargs` may refer to newly created or modified columns in `df`; items are computed and assigned into `df` in order. For Python 3.5 and below, the order of keyword arguments is not specified, you cannot refer to newly created or modified columns. All items are computed first, and then assigned in alphabetical order.

Changed in version 0.23.0: Keyword argument order is maintained for Python 3.6 and later.

**Examples**

```
>>> df = pd.DataFrame({'A': range(1, 11), 'B': np.random.randn(10)})
```

Where the value is a callable, evaluated on `df`:

```
>>> df.assign(ln_A = lambda x: np.log(x.A))
   A         B      ln_A
0  1  0.426905  0.000000
1  2 -0.780949  0.693147
2  3 -0.418711  1.098612
3  4 -0.269708  1.386294
4  5 -0.274002  1.609438
5  6 -0.500792  1.791759
6  7  1.649697  1.945910
7  8 -1.495604  2.079442
8  9  0.549296  2.197225
9 10 -0.758542  2.302585
```

Where the value already exists and is inserted:

```
>>> newcol = np.log(df['A'])
>>> df.assign(ln_A=newcol)
   A         B      ln_A
0  1  0.426905  0.000000
1  2 -0.780949  0.693147
2  3 -0.418711  1.098612
3  4 -0.269708  1.386294
4  5 -0.274002  1.609438
5  6 -0.500792  1.791759
6  7  1.649697  1.945910
7  8 -1.495604  2.079442
8  9  0.549296  2.197225
9 10 -0.758542  2.302585
```

Where the keyword arguments depend on each other

```
>>> df = pd.DataFrame({'A': [1, 2, 3]})
```

```
>>> df.assign(B=df.A, C=lambda x:x['A']+ x['B'])
   A  B  C
0  1  1  2
1  2  2  4
2  3  3  6
```

## pandas.DataFrame.astype

`DataFrame.astype(dtype, copy=True, errors='raise', **kwargs)`

Cast a pandas object to a specified dtype dtype.

**Parameters** **dtype** : data type, or dict of column name -> data type

Use a `numpy.dtype` or Python type to cast entire pandas object to the same type. Alternatively, use `{col: dtype, ...}`, where `col` is a column label and `dtype` is a `numpy.dtype` or Python type to cast one or more of the DataFrame's columns to column-specific types.

**copy** : bool, default True.

Return a copy when `copy=True` (be very careful setting `copy=False` as changes to values then may propagate to other pandas objects).

**errors** : {'raise', 'ignore'}, default 'raise'.

Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

**raise\_on\_error** : raise on invalid input

Deprecated since version 0.20.0: Use `errors` instead

**kwargs** [keyword arguments to pass on to the constructor]

### Returns

**casted** [type of caller]

See also:

[`pandas.to\_datetime`](#) Convert argument to datetime.

[`pandas.to\_timedelta`](#) Convert argument to timedelta.

[`pandas.to\_numeric`](#) Convert argument to a numeric type.

[`numpy.ndarray.astype`](#) Cast a numpy array to a specified type.

### Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1 # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

## pandas.DataFrame.at\_time

`DataFrame.at_time` (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

### Parameters

**time** [datetime.time or string]

### Returns

**values\_at\_time** [type of caller]

### Raises `TypeError`

If the index is not a *DatetimeIndex*

See also:

***between\_time*** Select values between particular times of the day

***first*** Select initial periods of time series based on a date offset



**last** Select final periods of time series based on a date offset

**DatetimeIndex.indexer\_at\_time** Get just the index locations for values at particular time of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09 00:00:00	1
2018-04-09 12:00:00	2
2018-04-10 00:00:00	3
2018-04-10 12:00:00	4

```
>>> ts.at_time('12:00')
```

	A
2018-04-09 12:00:00	2
2018-04-10 12:00:00	4

## pandas.DataFrame.between\_time

**DataFrame.between\_time** (*start\_time*, *end\_time*, *include\_start=True*, *include\_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting *start\_time* to be later than *end\_time*, you can get the times that are *not* between the two times.

### Parameters

**start\_time** [datetime.time or string]

**end\_time** [datetime.time or string]

**include\_start** [boolean, default True]

**include\_end** [boolean, default True]

### Returns

**values\_between\_time** [type of caller]

### Raises TypeError

If the index is not a *DatetimeIndex*

### See also:

**at\_time** Select values at a particular time of the day

**first** Select initial periods of time series based on a date offset

**last** Select final periods of time series based on a date offset

**DatetimeIndex.indexer\_between\_time** Get just the index locations for values between particular times of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09 00:00:00	1
2018-04-10 00:20:00	2
2018-04-11 00:40:00	3
2018-04-12 01:00:00	4

```
>>> ts.between_time('0:15', '0:45')
A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

## pandas.DataFrame.bfill

`DataFrame.bfill` (*axis=None, inplace=False, limit=None, downcast=None*)  
 Synonym for `DataFrame.fillna(method='bfill')`

## pandas.DataFrame.bool

`DataFrame.bool()`  
 Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a `ValueError` if the PandasObject does not have exactly 1 element, or that element is not boolean

## pandas.DataFrame.boxplot

`DataFrame.boxplot` (*column=None, by=None, ax=None, fontsize=None, rot=0, grid=True, figsize=None, layout=None, return\_type=None, \*\*kwargs*)  
 Make a box plot from DataFrame columns.

Make a box-and-whisker plot from DataFrame columns, optionally grouped by some other columns. A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to  $1.5 * IQR$  ( $IQR = Q3 - Q1$ ) from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for `boxplot`.

**Parameters** `column` : str or list of str, optional

Column name or list of names, or vector. Can be any valid input to `pandas.DataFrame.groupby()`.

**by** : str or array-like, optional

Column in the DataFrame to `pandas.DataFrame.groupby()`. One box-plot will be done per value of columns in *by*.

**ax** : object of class `matplotlib.axes.Axes`, optional

The matplotlib axes to be used by boxplot.

**fontsize** : float or str

Tick label font size in points or as a string (e.g., *large*).

**rot** : int or float, default 0

The rotation angle of labels (in degrees) with respect to the screen coordinate system.

**grid** : boolean, default True

Setting this to True will show the grid.

**figsize** : A tuple (width, height) in inches

The size of the figure to create in matplotlib.

**layout** : tuple (rows, columns), optional

For example, (3, 5) will display the subplots using 3 columns and 5 rows, starting from the top-left.

**return\_type** : { 'axes', 'dict', 'both' } or None, default 'axes'

The kind of object to return. The default is `axes`.

- 'axes' returns the matplotlib axes the boxplot is drawn on.
- 'dict' returns a dictionary whose values are the matplotlib Lines of the boxplot.
- 'both' returns a namedtuple with the axes and dict.
- when grouping with *by*, a Series mapping columns to *return\_type* is returned.

If *return\_type* is *None*, a NumPy array of axes with the same shape as *layout* is returned.

**\*\*kwds**

All other plotting keyword arguments to be passed to `matplotlib.pyplot.boxplot()`.

#### Returns result :

The return type depends on the *return\_type* parameter:

- 'axes' : object of class `matplotlib.axes.Axes`
- 'dict' : dict of `matplotlib.lines.Line2D` objects
- 'both' : a namedtuple with structure (ax, lines)

For data grouped with *by*:

- *Series*
- array (for *return\_type* = None)

See also:

`Series.plot.hist` Make a histogram.

`matplotlib.pyplot.boxplot` Matplotlib equivalent plot.

## Notes

Use `return_type='dict'` when you want to tweak the appearance of the lines after plotting. In this case a dict containing the Lines making up the boxes, caps, fliers, medians, and whiskers is returned.

## Examples

Boxplots can be created for every column in the dataframe by `df.boxplot()` or indicating the columns to be used:

```
>>> np.random.seed(1234)
>>> df = pd.DataFrame(np.random.randn(10,4),
...                    columns=['Col1', 'Col2', 'Col3', 'Col4'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2', 'Col3'])
```

Boxplots of variables distributions grouped by the values of a third variable can be created using the option `by`. For instance:

```
>>> df = pd.DataFrame(np.random.randn(10, 2),
...                    columns=['Col1', 'Col2'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                     'B', 'B', 'B', 'B', 'B'])
>>> boxplot = df.boxplot(by='X')
```

A list of strings (i.e. `['X', 'Y']`) can be passed to `boxplot` in order to group the data by combination of the variables in the x-axis:

```
>>> df = pd.DataFrame(np.random.randn(10,3),
...                    columns=['Col1', 'Col2', 'Col3'])
>>> df['X'] = pd.Series(['A', 'A', 'A', 'A', 'A',
...                     'B', 'B', 'B', 'B', 'B'])
>>> df['Y'] = pd.Series(['A', 'B', 'A', 'B', 'A',
...                     'B', 'A', 'B', 'A', 'B'])
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by=['X', 'Y'])
```

The layout of boxplot can be adjusted giving a tuple to `layout`:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                       layout=(2, 1))
```

Additional formatting can be done to the boxplot, like suppressing the grid (`grid=False`), rotating the labels in the x-axis (i.e. `rot=45`) or changing the fontsize (i.e. `fontsize=15`):

```
>>> boxplot = df.boxplot(grid=False, rot=45, fontsize=15)
```

The parameter `return_type` can be used to select the type of element returned by `boxplot`. When `return_type='axes'` is selected, the matplotlib axes on which the boxplot is drawn are returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], return_type='axes')
>>> type(boxplot)
<class 'matplotlib.axes._subplots.AxesSubplot'>
```

When grouping with `by`, a Series mapping columns to `return_type` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                        return_type='axes')
>>> type(boxplot)
<class 'pandas.core.series.Series'>
```

If `return_type` is `None`, a NumPy array of axes with the same shape as `layout` is returned:

```
>>> boxplot = df.boxplot(column=['Col1', 'Col2'], by='X',
...                        return_type=None)
>>> type(boxplot)
<class 'numpy.ndarray'>
```

## pandas.DataFrame.clip

`DataFrame.clip` (*lower=None, upper=None, axis=None, inplace=False, \*args, \*\*kwargs*)

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

**Parameters** **lower** : float or array\_like, default None

Minimum threshold value. All values below this threshold will be set to it.

**upper** : float or array\_like, default None

Maximum threshold value. All values above this threshold will be set to it.

**axis** : int or string axis name, optional

Align object with lower and upper along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data.

New in version 0.21.0.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns** **Series or DataFrame**

Same type as calling object with the values outside the clip boundaries replaced

**See also:**

**`clip_lower`** Clip values below specified threshold(s).

**`clip_upper`** Clip values above specified threshold(s).

## Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9    -2
1     -3    -7
2      0     6
3     -1     8
4      5    -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6    -2
1     -3    -4
2      0     6
3     -1     6
4      5    -4
```

Clips using specific lower and upper thresholds per column element:

```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0      2
1     -4
2     -1
3      6
4      3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

## pandas.DataFrame.clip\_lower

`DataFrame.clip_lower(threshold, axis=None, inplace=False)`

Return copy of the input with values below a threshold truncated.

**Parameters** `threshold` : numeric or array-like

Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for *axis=None*, or 1-D and the same length as the axis being compared.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Align *self* with *threshold* along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data.

New in version 0.21.0.

### Returns

**clipped** [same type as input]

### See also:

[`Series.clip`](#) Return copy of input with values below and above thresholds truncated.

[`Series.clip\_upper`](#) Return copy of input with values above threshold truncated.

### Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0      8
1      8
2      8
3      8
4      9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0      5
1      8
2      7
3      8
4      9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
1  4  5
2  5  6
```

## pandas.DataFrame.clip\_upper

`DataFrame.clip_upper` (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

### Parameters

**threshold** [float or array\_like]

**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

New in version 0.21.0.

### Returns

**clipped** [same type as input]

**See also:**

[\*clip\*](#)

## pandas.DataFrame.combine

`DataFrame.combine` (*other*, *func*, *fill\_value=None*, *overwrite=True*)

Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)

### Parameters

**other** [DataFrame]

**func** : function



Function that takes two series as inputs and return a Series or a scalar

**fill\_value** [scalar value]

**overwrite** : boolean, default True

If True then overwrite values for common keys in the calling frame

### Returns

**result** [DataFrame]

### See also:

**DataFrame.combine\_first** Combine two DataFrame objects and default to non-null values in frame calling the method

### Examples

```
>>> df1 = DataFrame({'A': [0, 0], 'B': [4, 4]})
>>> df2 = DataFrame({'A': [1, 1], 'B': [3, 3]})
>>> df1.combine(df2, lambda s1, s2: s1 if s1.sum() < s2.sum() else s2)
   A  B
0  0  3
1  0  3
```

## pandas.DataFrame.combine\_first

**DataFrame.combine\_first** (*other*)

Combine two DataFrame objects and default to non-null values in frame calling the method. Result index columns will be the union of the respective indexes and columns

### Parameters

**other** [DataFrame]

### Returns

**combined** [DataFrame]

### See also:

**DataFrame.combine** Perform series-wise operation on two DataFrames using a given function

### Examples

df1's values prioritized, use values from df2 to fill holes:

```
>>> df1 = pd.DataFrame([[1, np.nan]])
>>> df2 = pd.DataFrame([[3, 4]])
>>> df1.combine_first(df2)
   0  1
0  1  4.0
```

### pandas.DataFrame.compound

`DataFrame.compound` (*axis=None, skipna=None, level=None*)

Return the compound percentage of the values for the requested axis

#### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**compounded** [Series or DataFrame (if level specified)]

### pandas.DataFrame.consolidate

`DataFrame.consolidate` (*inplace=False*)

Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

### pandas.DataFrame.convert\_objects

`DataFrame.convert_objects` (*convert\_dates=True, convert\_numeric=False, convert\_timedeltas=True, copy=True*)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

**Parameters** **convert\_dates** : boolean, default True

If True, convert to date where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.

**convert\_numeric** : boolean, default False

If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

**convert\_timedeltas** : boolean, default True

If True, convert to timedelta where possible. If ‘coerce’, force conversion, with unconvertible values becoming NaT.

**copy** : boolean, default True

If True, return a copy even if no copy is necessary (e.g. no conversion was done).  
Note: This is meant for internal use, and should not be confused with inplace.

**Returns****converted** [same as input object]**See also:***pandas.to\_datetime* Convert argument to datetime.*pandas.to\_timedelta* Convert argument to timedelta.*pandas.to\_numeric* Return a fixed frequency timedelta index, with day as the default.**pandas.DataFrame.copy**`DataFrame.copy (deep=True)`

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

**Parameters** `deep` : bool, default True

Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

**Returns** `copy` : Series, DataFrame or Panel

Object type matches caller.

**Notes**

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to `copy.deepcopy` in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

**Examples**

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
```

(continues on next page)

(continued from previous page)

```
b      2
dtype: int64
```

**Shallow copy versus default (deep) copy:**

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a      3
b      4
dtype: int64
>>> shallow
a      3
b      4
dtype: int64
>>> deep
a      1
b      2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0      [10, 2]
1      [3, 4]
dtype: object
>>> deep
0      [10, 2]
1      [3, 4]
dtype: object
```

**pandas.DataFrame.corr**

`DataFrame.corr` (*method='pearson', min\_periods=1*)

Compute pairwise correlation of columns, excluding NA/null values

**Parameters** **method** : {'pearson', 'kendall', 'spearman'}

- **pearson** : standard correlation coefficient
- **kendall** : Kendall Tau correlation coefficient
- **spearman** : Spearman rank correlation

**min\_periods** : int, optional

Minimum number of observations required per pair of columns to have a valid result. Currently only available for pearson and spearman correlation

**Returns**

**y** [DataFrame]

**pandas.DataFrame.corrwith**

`DataFrame.corrwith` (*other, axis=0, drop=False*)

Compute pairwise correlation between rows or columns of two DataFrame objects.

**Parameters**

**other** [DataFrame, Series]

**axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

**drop** : boolean, default False

Drop missing indices from result, default returns union of all

**Returns**

**correls** [Series]

**pandas.DataFrame.count**

`DataFrame.count` (*axis=0, level=None, numeric\_only=False*)

Count non-NA cells for each column or row.

The values *None*, *NaN*, *NaT*, and optionally *numpy.inf* (depending on *pandas.options.mode.use\_inf\_as\_na*) are considered NA.

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

If 0 or 'index' counts are generated for each column. If 1 or 'columns' counts are generated for each **row**.

**level** : int or str, optional

If the axis is a *MultiIndex* (hierarchical), count along a particular *level*, collapsing into a *DataFrame*. A *str* specifies the level name.

**numeric\_only** : boolean, default False

Include only *float*, *int* or *boolean* data.

### Returns Series or DataFrame

For each column/row the number of non-NA/null entries. If *level* is specified returns a *DataFrame*.

### See also:

[\*Series.count\*](#) number of non-NA elements in a Series

[\*DataFrame.shape\*](#) number of DataFrame rows and columns (including NA elements)

[\*DataFrame.isna\*](#) boolean same-sized DataFrame showing places of NA elements

### Examples

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                     ["John", "Myla", None, "John", "Myla"],
...                     "Age": [24., np.nan, 21., 33, 26],
...                     "Single": [False, True, True, True, False]})
>>> df
   Person  Age  Single
0   John  24.0   False
1   Myla   NaN    True
2   None  21.0    True
3   John  33.0    True
4   Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person      4
Age         4
Single      5
dtype: int64
```

Counts for each row:

```
>>> df.count(axis='columns')
0      3
1      2
2      2
3      3
4      3
dtype: int64
```

Counts for one level of a *MultiIndex*:

```
>>> df.set_index(["Person", "Single"]).count(level="Person")
Age
Person
John      2
Myla      1
```

**pandas.DataFrame.cov**

`DataFrame.cov` (*min\_periods=None*)

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the [covariance matrix](#) of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as NaN.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

**Parameters** `min_periods` : int, optional

Minimum number of observations required per pair of columns to have a valid result.

**Returns** `DataFrame`

The covariance matrix of the series of the DataFrame.

**See also:**

[pandas.Series.cov](#) compute covariance with another Series

[pandas.core.window.EWM.cov](#) exponential weighted sample covariance

[pandas.core.window.Expanding.cov](#) expanding sample covariance

[pandas.core.window.Rolling.cov](#) rolling sample covariance

**Notes**

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is [missing at random](#)) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See [Estimation of covariance matrices](#) for more details.

**Examples**

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                    columns=['dogs', 'cats'])
>>> df.cov()
           dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```

>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                    columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()

```

	a	b	c	d	e
a	0.998438	-0.020161	0.059277	-0.008943	0.014144
b	-0.020161	1.059352	-0.008543	-0.024738	0.009826
c	0.059277	-0.008543	1.010670	-0.001486	-0.000271
d	-0.008943	-0.024738	-0.001486	0.921297	-0.013692
e	0.014144	0.009826	-0.000271	-0.013692	0.977795

### Minimum number of periods

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```

>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                    columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)

```

	a	b	c
a	0.316741	NaN	-0.150812
b	NaN	1.248003	0.191417
c	-0.150812	0.191417	0.895202

## pandas.DataFrame.cummax

`DataFrame.cummax` (*axis=None, skipna=True, \*args, \*\*kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

**Parameters** `axis` : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

`skipna` : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

`*args, **kwargs` :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

`cummax` [Series or DataFrame]

See also:

[`pandas.core.window.Expanding.max`](#) Similar functionality but ignores NaN values.

[`DataFrame.max`](#) Return the maximum over DataFrame axis.

[`DataFrame.cummax`](#) Return cumulative maximum over DataFrame axis.

[`DataFrame.cummin`](#) Return cumulative minimum over DataFrame axis.



**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
   A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
   A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

## pandas.DataFrame.cummin

`DataFrame.cummin` (*axis=None, skipna=True, \*args, \*\*kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs** :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**cummin** [Series or DataFrame]

See also:

[`pandas.core.window.Expanding.min`](#) Similar functionality but ignores NaN values.

[`DataFrame.min`](#) Return the minimum over DataFrame axis.

[`DataFrame.cummax`](#) Return cumulative maximum over DataFrame axis.

[`DataFrame.cummin`](#) Return cumulative minimum over DataFrame axis.

[`DataFrame.cumsum`](#) Return cumulative sum over DataFrame axis.

[`DataFrame.cumprod`](#) Return cumulative product over DataFrame axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A  B
0  2.0 1.0
1  3.0 NaN
2  1.0 0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
   A  B
0  2.0 1.0
1  2.0 NaN
2  1.0 0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
   A  B
0  2.0 1.0
1  3.0 NaN
2  1.0 0.0
```

### pandas.DataFrame.cumprod

`DataFrame.cumprod(axis=None, skipna=True, *args, **kwargs)`

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

**Parameters** `axis`: {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs** :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**cumprod** [Series or DataFrame]

See also:

**pandas.core.window.Expanding.prod** Similar functionality but ignores NaN values.

**DataFrame.prod** Return the product over DataFrame axis.

**DataFrame.cummax** Return cumulative maximum over DataFrame axis.

**DataFrame.cummin** Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0    2.0
1    NaN
2   10.0
3  -10.0
4   -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
   A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
   A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

## pandas.DataFrame.cumsum

`DataFrame.cumsum(axis=None, skipna=True, *args, **kwargs)`

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

**Parameters** `axis`: {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs**:

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**cumsum** [Series or DataFrame]

See also:

[`pandas.core.window.Expanding.sum`](#) Similar functionality but ignores NaN values.

[`DataFrame.sum`](#) Return the sum over DataFrame axis.

[`DataFrame.cummax`](#) Return cumulative maximum over DataFrame axis.

[`DataFrame.cummin`](#) Return cumulative minimum over DataFrame axis.

**DataFrame.cumsum** Return cumulative sum over DataFrame axis.

**DataFrame.cumprod** Return cumulative product over DataFrame axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
   A    B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
      A      B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

## pandas.DataFrame.describe

`DataFrame.describe` (*percentiles=None, include=None, exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters** `percentiles` : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

**include** : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use 'category'
- None (default) : The result will include all numeric columns.

**exclude** : list-like of dtypes or None (default), optional,

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use 'category'
- None (default) : The result will exclude nothing.

## Returns

**summary:** `Series/DataFrame` of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

## Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as `lower`, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

## Examples

Describing a numeric Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
top        a
freq       2
dtype: object
```

Describing a timestamp Series.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count      3
unique     2
top        2010-01-01 00:00:00
```

(continues on next page)



(continued from previous page)

```

freq                2
first      2000-01-01 00:00:00
last       2010-01-01 00:00:00
dtype: object

```

Describing a DataFrame. By default only numeric fields are returned.

```

>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                      'numeric': [1, 2, 3],
...                      'categorical': pd.Categorical(['d', 'e', 'f'])
...                      })
>>> df.describe()

```

	numeric
count	3.0
mean	2.0
std	1.0
min	1.0
25%	1.5
50%	2.0
75%	2.5
max	3.0

Describing all columns of a DataFrame regardless of data type.

```

>>> df.describe(include='all')

```

	categorical	numeric	object
count	3	3.0	3
unique	3	NaN	3
top	f	NaN	c
freq	1	NaN	1
mean	NaN	2.0	NaN
std	NaN	1.0	NaN
min	NaN	1.0	NaN
25%	NaN	1.5	NaN
50%	NaN	2.0	NaN
75%	NaN	2.5	NaN
max	NaN	3.0	NaN

Describing a column from a DataFrame by accessing it as an attribute.

```

>>> df.numeric.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max         3.0
Name: numeric, dtype: float64

```

Including only numeric columns in a DataFrame description.

```

>>> df.describe(include=[np.number])

```

	numeric
count	3.0

(continues on next page)

(continued from previous page)

mean	2.0
std	1.0
min	1.0
25%	1.5
50%	2.0
75%	2.5
max	3.0

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
      object
count      3
unique      3
top         c
freq        1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
      categorical
count           3
unique          3
top             f
freq            1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
      categorical  object
count           3      3
unique          3      3
top             f      c
freq            1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
      categorical  numeric
count           3      3.0
unique          3      NaN
top             f      NaN
freq            1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

## **pandas.DataFrame.diff**

DataFrame.**diff** (*periods=1, axis=0*)

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

**Parameters** `periods` : int, default 1

Periods to shift for calculating difference, accepts negative values.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Take difference over rows (0) or columns (1).

New in version 0.16.1..

**Returns**

**diffed** [DataFrame]

**See also:**

*[Series.diff](#)* First discrete difference for a Series.

*[DataFrame.pct\\_change](#)* Percent change over given number of periods.

*[DataFrame.shift](#)* Shift index by desired number of periods with an optional time freq.

## Examples

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b  c
0  1  1  1
1  2  1  4
2  3  2  9
3  4  3 16
4  5  5 25
5  6  8 36
```

```
>>> df.diff()
   a    b    c
0 NaN NaN NaN
1 1.0 0.0 3.0
2 1.0 1.0 5.0
3 1.0 1.0 7.0
4 1.0 2.0 9.0
5 1.0 3.0 11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
   a    b    c
0 NaN 0.0 0.0
1 NaN -1.0 3.0
2 NaN -1.0 7.0
3 NaN -1.0 13.0
4 NaN 0.0 20.0
5 NaN 2.0 28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
   a    b    c
0 NaN NaN NaN
1 NaN NaN NaN
2 NaN NaN NaN
3 3.0 2.0 15.0
4 3.0 4.0 21.0
5 3.0 6.0 27.0
```

Difference with following row

```
>>> df.diff(periods=-1)
   a    b    c
0 -1.0 0.0 -3.0
1 -1.0 -1.0 -5.0
2 -1.0 -1.0 -7.0
3 -1.0 -2.0 -9.0
4 -1.0 -3.0 -11.0
5 NaN NaN NaN
```

## pandas.DataFrame.div

`DataFrame.div` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

### Returns

**result** [DataFrame]

### See also:

`DataFrame.rtruediv`

### Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.divide

`DataFrame.divide` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

**See also:**

`DataFrame.rtruediv`

## Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.dot

`DataFrame.dot` (*other*)

Matrix multiplication with DataFrame or Series objects. Can also be called using *self @ other* in Python >= 3.5.

#### Parameters

**other** [DataFrame or Series]

#### Returns

**dot\_product** [DataFrame or Series]

## **pandas.DataFrame.drop**

`DataFrame.drop` (*labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise'*)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

**Parameters** **labels** : single label or list-like

Index or column labels to drop.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

**index, columns** : single label or list-like

Alternative to specifying axis (*labels, axis=1* is equivalent to *columns=labels*).

New in version 0.21.0.

**level** : int or level name, optional

For MultiIndex, level from which the labels will be removed.

**inplace** : bool, default False

If True, do operation inplace and return None.

**errors** : {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and only existing labels are dropped.

**Returns**

**dropped** [pandas.DataFrame]

**Raises** **KeyError**

If none of the labels are found in the selected axis

**See also:**

[\*DataFrame.loc\*](#) Label-location based indexer for selection by label.

[\*DataFrame.dropna\*](#) Return DataFrame with labels on given axis omitted where (all or any) data are missing

[\*DataFrame.drop\\_duplicates\*](#) Return DataFrame with duplicate rows removed, optionally only considering certain columns

[\*Series.drop\*](#) Return Series with specified index labels removed.

## **Examples**

```
>>> df = pd.DataFrame(np.arange(12).reshape(3,4),
...                    columns=['A', 'B', 'C', 'D'])
>>> df
   A  B  C  D
0  0  1  2  3
1  4  5  6  7
2  8  9 10 11
```

### Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A  D
0  0  3
1  4  7
2  8 11
```

```
>>> df.drop(columns=['B', 'C'])
   A  D
0  0  3
1  4  7
2  8 11
```

### Drop a row by index

```
>>> df.drop([0, 1])
   A  B  C  D
2  8  9 10 11
```

### Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                               ['speed', 'weight', 'length']],
...                       labels=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                               [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                    data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                          [250, 150], [1.5, 0.8], [320, 250],
...                          [1, 0.8], [0.3, 0.2]])
>>> df
```

		big	small
lama	speed	45.0	30.0
	weight	200.0	100.0
	length	1.5	1.0
cow	speed	30.0	20.0
	weight	250.0	150.0
	length	1.5	0.8
falcon	speed	320.0	250.0
	weight	1.0	0.8
	length	0.3	0.2

```
>>> df.drop(index='cow', columns='small')
           big
lama  speed  45.0
      weight 200.0
      length  1.5
falcon speed 320.0
```

(continues on next page)

(continued from previous page)

weight	1.0
length	0.3

```
>>> df.drop(index='length', level=1)
```

		big	small
lama	speed	45.0	30.0
	weight	200.0	100.0
cow	speed	30.0	20.0
	weight	250.0	150.0
falcon	speed	320.0	250.0
	weight	1.0	0.8

## pandas.DataFrame.drop\_duplicates

`DataFrame.drop_duplicates` (*subset=None, keep='first', inplace=False*)

Return DataFrame with duplicate rows removed, optionally only considering certain columns

**Parameters** **subset** : column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns

**keep** : {'first', 'last', False}, default 'first'

- **first** : Drop duplicates except for the first occurrence.
- **last** : Drop duplicates except for the last occurrence.
- **False** : Drop all duplicates.

**inplace** : boolean, default False

Whether to drop duplicates in place or to return a copy

**Returns**

**deduplicated** [DataFrame]

## pandas.DataFrame.dropna

`DataFrame.dropna` (*axis=0, how='any', thresh=None, subset=None, inplace=False*)

Remove missing values.

See the [User Guide](#) for more on which values are considered missing, and how to work with missing data.

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

Determine if rows or columns which contain missing values are removed.

- 0, or 'index' : Drop rows which contain missing values.
- 1, or 'columns' : Drop columns which contain missing value.

Deprecated since version 0.23.0:: Pass tuple or list to drop on multiple axes.

**how** : {'any', 'all'}, default 'any'



Determine if row or column is removed from DataFrame, when we have at least one NA or all NA.

- ‘any’ : If any NA values are present, drop that row or column.
- ‘all’ : If all values are NA, drop that row or column.

**thresh** : int, optional

Require that many non-NA values.

**subset** : array-like, optional

Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include.

**inplace** : bool, default False

If True, do operation inplace and return None.

### Returns DataFrame

DataFrame with NA entries dropped from it.

See also:

[`DataFrame.isna`](#) Indicate missing values.

[`DataFrame.notna`](#) Indicate existing (non-missing) values.

[`DataFrame.fillna`](#) Replace missing values.

[`Series.dropna`](#) Drop missing values.

[`Index.dropna`](#) Drop missing indices.

### Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                     "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                     "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                               pd.NaT]})
>>> df
```

	name	toy	born
0	Alfred	NaN	NaT
1	Batman	Batmobile	1940-04-25
2	Catwoman	Bullwhip	NaT

Drop the rows where at least one element is missing.

```
>>> df.dropna()
```

	name	toy	born
1	Batman	Batmobile	1940-04-25

Drop the columns where at least one element is missing.

```
>>> df.dropna(axis='columns')
```

	name
0	Alfred
1	Batman
2	Catwoman

Drop the rows where all elements are missing.

```
>>> df.dropna(how='all')
   name      toy      born
0  Alfred    NaN     NaT
1  Batman  Batmobile 1940-04-25
2 Catwoman  Bullwhip     NaT
```

Keep only the rows with at least 2 non-NA values.

```
>>> df.dropna(thresh=2)
   name      toy      born
1  Batman  Batmobile 1940-04-25
2 Catwoman  Bullwhip     NaT
```

Define in which columns to look for missing values.

```
>>> df.dropna(subset=['name', 'born'])
   name      toy      born
1  Batman  Batmobile 1940-04-25
```

Keep the DataFrame with valid entries in the same variable.

```
>>> df.dropna(inplace=True)
>>> df
   name      toy      born
1  Batman  Batmobile 1940-04-25
```

## pandas.DataFrame.duplicated

`DataFrame.duplicated` (*subset=None, keep='first'*)

Return boolean Series denoting duplicate rows, optionally only considering certain columns

**Parameters** **subset** : column label or sequence of labels, optional

Only consider certain columns for identifying duplicates, by default use all of the columns

**keep** : {'first', 'last', False}, default 'first'

- **first** : Mark duplicates as True except for the first occurrence.
- **last** : Mark duplicates as True except for the last occurrence.
- **False** : Mark all duplicates as True.

**Returns**

**duplicated** [Series]

## pandas.DataFrame.eq

`DataFrame.eq` (*other, axis='columns', level=None*)

Wrapper for flexible comparison methods eq

**pandas.DataFrame.equals**`DataFrame.equals` (*other*)

Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

**pandas.DataFrame.eval**`DataFrame.eval` (*expr*, *inplace=False*, *\*\*kwargs*)

Evaluate a string describing operations on DataFrame columns.

Operates on columns only, not specific rows or elements. This allows *eval* to run arbitrary code, which can make you vulnerable to code injection if you pass user input to this function.

**Parameters** *expr* : str

The expression string to evaluate.

**inplace** : bool, default False

If the expression contains an assignment, whether to perform the operation in-place and mutate the existing DataFrame. Otherwise, a new DataFrame is returned.

New in version 0.18.0..

**kwargs** : dict

See the documentation for `eval()` for complete details on the keyword arguments accepted by `query()`.

**Returns** ndarray, scalar, or pandas object

The result of the evaluation.

**See also:**

**`DataFrame.query`** Evaluates a boolean expression to query the columns of a frame.

**`DataFrame.assign`** Can evaluate an expression or function to create new values for a column.

**`pandas.eval`** Evaluate a Python expression as a string using various backends.

**Notes**

For more details see the API documentation for `eval()`. For detailed examples see *enhancing performance with eval*.

**Examples**

```
>>> df = pd.DataFrame({'A': range(1, 6), 'B': range(10, 0, -2)})
>>> df
   A  B
0  1 10
1  2  8
2  3  6
3  4  4
```

(continues on next page)

(continued from previous page)

```

4  5    2
>>> df.eval('A + B')
0     11
1     10
2      9
3      8
4      7
dtype: int64

```

Assignment is allowed though by default the original DataFrame is not modified.

```

>>> df.eval('C = A + B')
   A  B  C
0  1 10 11
1  2  8 10
2  3  6  9
3  4  4  8
4  5  2  7
>>> df
   A  B
0  1 10
1  2  8
2  3  6
3  4  4
4  5  2

```

Use `inplace=True` to modify the original DataFrame.

```

>>> df.eval('C = A + B', inplace=True)
>>> df
   A  B  C
0  1 10 11
1  2  8 10
2  3  6  9
3  4  4  8
4  5  2  7

```

## pandas.DataFrame.ewm

`DataFrame.ewm(com=None, span=None, halflife=None, alpha=None, min_periods=0, adjust=True, ignore_na=False, axis=0)`

Provides exponential weighted functions

New in version 0.18.0.

**Parameters** `com` : float, optional

Specify decay in terms of center of mass,  $\alpha = 1/(1 + com)$ , for  $com \geq 0$

`span` : float, optional

Specify decay in terms of span,  $\alpha = 2/(span + 1)$ , for  $span \geq 1$

`halflife` : float, optional

Specify decay in terms of half-life,  $\alpha = 1 - \exp(\log(0.5)/halflife)$ , for  $halflife > 0$

**alpha** : float, optional

Specify smoothing factor  $\alpha$  directly,  $0 < \alpha \leq 1$

New in version 0.18.0.

**min\_periods** : int, default 0

Minimum number of observations in window required to have a value (otherwise result is NA).

**adjust** : boolean, default True

Divide by decaying adjustment factor in beginning periods to account for imbalance in relative weightings (viewing EWMA as a moving average)

**ignore\_na** : boolean, default False

Ignore missing values when calculating weights; specify True to reproduce pre-0.15.0 behavior

### Returns

**a Window sub-classed for the particular operation**

See also:

*rolling* Provides rolling window calculations

*expanding* Provides expanding transformations.

### Notes

Exactly one of center of mass, span, half-life, and alpha must be provided. Allowed values and relationship between the parameters are specified in the parameter descriptions above; see the link at the end of this section for a detailed explanation.

When adjust is True (default), weighted averages are calculated using weights  $(1-\alpha)^{(n-1)}$ ,  $(1-\alpha)^{(n-2)}$ , ...,  $1-\alpha$ , 1.

**When adjust is False, weighted averages are calculated recursively as:**  $\text{weighted\_average}[0] = \text{arg}[0]$ ;  $\text{weighted\_average}[i] = (1-\alpha) * \text{weighted\_average}[i-1] + \alpha * \text{arg}[i]$ .

When ignore\_na is False (default), weights are based on absolute positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are  $(1-\alpha)^2$  and 1 (if adjust is True), and  $(1-\alpha)^2$  and alpha (if adjust is False).

When ignore\_na is True (reproducing pre-0.15.0 behavior), weights are based on relative positions. For example, the weights of x and y used in calculating the final weighted average of [x, None, y] are  $1-\alpha$  and 1 (if adjust is True), and  $1-\alpha$  and alpha (if adjust is False).

More details can be found at <http://pandas.pydata.org/pandas-docs/stable/computation.html#exponentially-weighted-windows>

### Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
      B
0  0.0
1  1.0
```

(continues on next page)

(continued from previous page)

```
2  2.0
3  NaN
4  4.0
```

```
>>> df.ewm(com=0.5).mean()
      B
0  0.000000
1  0.750000
2  1.615385
3  1.615385
4  3.670213
```

## pandas.DataFrame.expanding

`DataFrame.expanding` (*min\_periods=1, center=False, axis=0*)

Provides expanding transformations.

New in version 0.18.0.

**Parameters** `min_periods` : int, default 1

Minimum number of observations in window required to have a value (otherwise result is NA).

**center** : boolean, default False

Set the labels at the center of the window.

**axis** [int or string, default 0]

### Returns

a Window sub-classed for the particular operation

See also:

[\*rolling\*](#) Provides rolling window calculations

[\*ewm\*](#) Provides exponential weighted functions

### Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

### Examples

```
>>> df = DataFrame({'B': [0, 1, 2, np.nan, 4]})
      B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

```
>>> df.expanding(2).sum()
      B
0  NaN
1  1.0
2  3.0
3  3.0
4  7.0
```

### pandas.DataFrame.fffll

DataFrame.fffll (*axis=None, inplace=False, limit=None, downcast=None*)

Synonym for `DataFrame.fillna(method='fffll')`

### pandas.DataFrame.fillna

DataFrame.fillna (*value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, \*\*kwargs*)

Fill NA/NaN values using the specified method

**Parameters** **value** : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

**method** : {'backfill', 'bfill', 'pad', 'fffll', None}, default None

Method to use for filling holes in reindexed Series pad / ffffll: propagate last valid observation forward to next valid backfill / bfffll: use NEXT valid observation to fill gap

**axis** [{0 or 'index', 1 or 'columns'}]

**inplace** : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

**limit** : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

**downcast** : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

#### Returns

**filled** [DataFrame]

See also:

*interpolate* Fill NaN values using interpolation.

*reindex, asfreq*

## Examples

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                     [3, 4, np.nan, 1],
...                     [np.nan, np.nan, np.nan, 5],
...                     [np.nan, 3, np.nan, 4]],
...                     columns=list('ABCD'))
>>> df
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	NaN	4

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
```

	A	B	C	D
0	0.0	2.0	0.0	0
1	3.0	4.0	0.0	1
2	0.0	0.0	0.0	5
3	0.0	3.0	0.0	4

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
```

	A	B	C	D
0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	3.0	4.0	NaN	5
3	3.0	3.0	NaN	4

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
```

	A	B	C	D
0	0.0	2.0	2.0	0
1	3.0	4.0	2.0	1
2	0.0	1.0	2.0	5
3	0.0	3.0	2.0	4

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
```

	A	B	C	D
0	0.0	2.0	2.0	0
1	3.0	4.0	NaN	1
2	NaN	1.0	NaN	5
3	NaN	3.0	NaN	4



**pandas.DataFrame.filter**

`DataFrame.filter` (*items=None, like=None, regex=None, axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

**Parameters** *items* : list-like

List of info axis to restrict to (must not all be present)

**like** : string

Keep info axis where “arg in col == True”

**regex** : string (regular expression)

Keep info axis with `re.search(regex, col) == True`

**axis** : int or string axis name

The axis to filter on. By default this is the info axis, ‘index’ for Series, ‘columns’ for DataFrame

**Returns**

same type as input object

**See also:**

`pandas.DataFrame.loc`

**Notes**

The *items*, *like*, and *regex* parameters are enforced to be mutually exclusive.

*axis* defaults to the info axis that is used when indexing with `[]`.

**Examples**

```
>>> df
one two three
mouse 1 2 3
rabbit 4 5 6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one three
mouse 1 3
rabbit 4 6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one three
mouse 1 3
rabbit 4 6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one two three
rabbit 4 5 6
```

## pandas.DataFrame.first

`DataFrame.first(offset)`

Convenience method for subsetting initial periods of time series data based on a date offset.

### Parameters

**offset** [string, DateOffset, dateutil.relativedelta]

### Returns

**subset** [type of caller]

### Raises `TypeError`

If the index is not a *DatetimeIndex*

### See also:

**last** Select final periods of time series based on a date offset

**at\_time** Select values at a particular time of the day

**between\_time** Select values between particular times of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
           A
2018-04-09  1
2018-04-11  2
2018-04-13  3
2018-04-15  4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
           A
2018-04-09  1
2018-04-11  2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

## pandas.DataFrame.first\_valid\_index

`DataFrame.first_valid_index()`

Return index for first non-NA/null value.

**Returns****scalar** [type of index]**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

**pandas.DataFrame.floordiv**

`DataFrame.floordiv` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Integer division of dataframe and other, element-wise (binary operator *floordiv*).

Equivalent to `dataframe // other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

**Parameters**

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

**Returns****result** [DataFrame]**See also:**

`DataFrame.rfloordiv`

**Notes**

Mismatched indices will be unioned together

**Examples**

None

**pandas.DataFrame.from\_csv**

**classmethod** `DataFrame.from_csv` (*path*, *header*=0, *sep*=' ', *index\_col*=0, *parse\_dates*=True, *encoding*=None, *tupleize\_cols*=None, *infer\_datetime\_format*=False)

Read CSV file.

Deprecated since version 0.21.0: Use `pandas.read_csv()` instead.

It is preferable to use the more powerful `pandas.read_csv()` for most general purposes, but `from_csv` makes for an easy roundtrip to and from a file (the exact counterpart of `to_csv`), especially with a DataFrame of time series data.

This method only differs from the preferred `pandas.read_csv()` in some defaults:

- `index_col` is 0 instead of `None` (take first column as index by default)
- `parse_dates` is `True` instead of `False` (try parsing the index as datetime by default)

So a `pd.DataFrame.from_csv(path)` can be replaced by `pd.read_csv(path, index_col=0, parse_dates=True)`.

#### Parameters

**path** [string file path or file handle / StringIO]

**header** : int, default 0

Row to use as header (skip prior rows)

**sep** : string, default ','

Field delimiter

**index\_col** : int or sequence, default 0

Column to use for index. If a sequence is given, a MultiIndex is used. Different default from `read_table`

**parse\_dates** : boolean, default True

Parse dates. Different default from `read_table`

**tupleize\_cols** : boolean, default False

write multi\_index columns as a list of tuples (if True) or new (expanded format) if False)

**infer\_datetime\_format**: boolean, default False

If True and `parse_dates` is True for a column, try to infer the datetime format based on the first datetime string. If the format can be inferred, there often will be a large parsing speed-up.

#### Returns

**y** [DataFrame]

See also:

`pandas.read_csv`

### **pandas.DataFrame.from\_dict**

**classmethod** `DataFrame.from_dict` (*data*, *orient*='columns', *dtype*=None, *columns*=None)  
Construct DataFrame from dict of array-like or dicts.

Creates DataFrame object from dictionary by columns or by index allowing dtype specification.

**Parameters** **data** : dict

Of the form {field : array-like} or {field : dict}.

**orient** : {'columns', 'index'}, default 'columns'

The “orientation” of the data. If the keys of the passed dict should be the columns of the resulting DataFrame, pass 'columns' (default). Otherwise if the keys should be rows, pass 'index'.

**dtype** : dtype, default None

Data type to force, otherwise infer.

**columns** : list, default None

Column labels to use when orient='index'. Raises a ValueError if used with orient='columns'.

New in version 0.23.0.

### Returns

**pandas.DataFrame**

See also:

**DataFrame.from\_records** DataFrame from ndarray (structured dtype), list of tuples, dict, or DataFrame

**DataFrame** DataFrame object creation using constructor

### Examples

By default the keys of the dict become the DataFrame columns:

```
>>> data = {'col_1': [3, 2, 1, 0], 'col_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data)
   col_1 col_2
0      3    a
1      2    b
2      1    c
3      0    d
```

Specify orient='index' to create the DataFrame using dictionary keys as rows:

```
>>> data = {'row_1': [3, 2, 1, 0], 'row_2': ['a', 'b', 'c', 'd']}
>>> pd.DataFrame.from_dict(data, orient='index')
      0  1  2  3
row_1  3  2  1  0
row_2  a  b  c  d
```

When using the 'index' orientation, the column names can be specified manually:

```
>>> pd.DataFrame.from_dict(data, orient='index',
...                          columns=['A', 'B', 'C', 'D'])
      A  B  C  D
row_1  3  2  1  0
row_2  a  b  c  d
```

### pandas.DataFrame.from\_items

**classmethod** `DataFrame.from_items` (*items*, *columns=None*, *orient='columns'*)

Construct a dataframe from a list of tuples

Deprecated since version 0.23.0: `from_items` is deprecated and will be removed in a future version. Use `DataFrame.from_dict(dict(items))` instead. `DataFrame.from_dict(OrderedDict(items))` may be used to preserve the key order.

Convert (key, value) pairs to DataFrame. The keys will be the axis index (usually the columns, but depends on the specified orientation). The values should be arrays or Series.

**Parameters** *items* : sequence of (key, value) pairs

Values should be arrays or Series.

**columns** : sequence of column labels, optional

Must be passed if *orient*='index'.

**orient** : {'columns', 'index'}, default 'columns'

The “orientation” of the data. If the keys of the input correspond to column labels, pass 'columns' (default). Otherwise if the keys correspond to the index, pass 'index'.

**Returns**

**frame** [DataFrame]

### pandas.DataFrame.from\_records

**classmethod** `DataFrame.from_records` (*data*, *index=None*, *exclude=None*, *columns=None*, *coerce\_float=False*, *nrows=None*)

Convert structured or record ndarray to DataFrame

**Parameters**

**data** [ndarray (structured dtype), list of tuples, dict, or DataFrame]

**index** : string, list of fields, array-like

Field of array to use as the index, alternately a specific set of input labels to use

**exclude** : sequence, default None

Columns or fields to exclude

**columns** : sequence, default None

Column names to use. If the passed data do not have names associated with them, this argument provides names for the columns. Otherwise this argument indicates the order of the columns in the result (any names not found in the data will become all-NA columns)

**coerce\_float** : boolean, default False

Attempt to convert values of non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

**Returns**

**df** [DataFrame]

**pandas.DataFrame.ge**`DataFrame.ge` (*other*, *axis*='columns', *level*=None)Wrapper for flexible comparison methods `ge`**pandas.DataFrame.get**`DataFrame.get` (*key*, *default*=None)

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

**Parameters****key** [object]**Returns****value** [type of items contained in object]**pandas.DataFrame.get\_dtype\_counts**`DataFrame.get_dtype_counts` ()

Return counts of unique dtypes in this object.

**Returns dtype** : Series

Series with the count of columns with each dtype.

**See also:**[`dtypes`](#) Return the dtypes in this object.**Examples**

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
   str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```

**pandas.DataFrame.get\_ftype\_counts**`DataFrame.get_ftype_counts` ()

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for `SparseDataFrame` or for `DataFrames` containing sparse arrays.

**Returns** `dtype` : Series

Series with the count of columns with each type and sparsity (dense/sparse)

**See also:**

*ftypes* Return ftypes (indication of sparse/dense and dtype) in this object.

### Examples

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
   str  int  float
0   a    1   1.0
1   b    2   2.0
2   c    3   3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

### `pandas.DataFrame.get_value`

`DataFrame.get_value(index, col, takeable=False)`

Quickly retrieve single value at passed column and index

Deprecated since version 0.21.0: Use `.at[]` or `.iat[]` accessors instead.

#### Parameters

**index** [row label]

**col** [column label]

**takeable** [interpret the index/col as indexers, default False]

#### Returns

**value** [scalar value]

### `pandas.DataFrame.get_values`

`DataFrame.get_values()`

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a `pandas.SparseArray`, the data are first converted to a dense representation.

**Returns** `numpy.ndarray`

Numpy representation of `DataFrame`

**See also:**



**values** Numpy representation of DataFrame.

**pandas.SparseArray** Container for sparse data.

## Examples

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a     b     c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
   a     c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

## pandas.DataFrame.groupby

**DataFrame.groupby** (*by=None, axis=0, level=None, as\_index=True, sort=True, group\_keys=True, squeeze=False, observed=False, \*\*kwargs*)

Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.

**Parameters** **by** : mapping, function, label, or list of labels

Used to determine the groups for the groupby. If *by* is a function, it's called on each value of the object's index. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups (the Series' values are first aligned; see `.align()` method). If an ndarray is passed, the values are used as-is determine the groups. A label or list of labels may be passed to group by the columns in *self*. Notice that a tuple is interpreted a (single) key.

**axis** [int, default 0]

**level** : int, level name, or sequence of such, default None

If the axis is a MultiIndex (hierarchical), group by a particular level or levels

**as\_index** : boolean, default True

For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. *as\_index=False* is effectively "SQL-style" grouped output

**sort** : boolean, default True

Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. `groupby` preserves the order of rows within each group.

**group\_keys** : boolean, default True

When calling `apply`, add group keys to index to identify pieces

**squeeze** : boolean, default False

reduce the dimensionality of the return type if possible, otherwise return a consistent type

**observed** : boolean, default False

This only applies if any of the groupers are Categoricals. If True: only show observed values for categorical groupers. If False: show all values for categorical groupers.

New in version 0.23.0.

## Returns

### GroupBy object

See also:

[\*resample\*](#) Convenience method for frequency conversion and resampling of time series.

## Notes

See the [user guide](#) for more.

## Examples

DataFrame results

```
>>> data.groupby(func, axis=0).mean()
>>> data.groupby(['col1', 'col2'])['col3'].mean()
```

DataFrame with hierarchical index

```
>>> data.groupby(['col1', 'col2']).mean()
```

## pandas.DataFrame.gt

`DataFrame.gt` (*other*, *axis*='columns', *level*=None)  
Wrapper for flexible comparison methods `gt`

## pandas.DataFrame.head

`DataFrame.head` (*n*=5)  
Return the first *n* rows.

This function returns the first  $n$  rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

**Parameters** **n** : int, default 5

Number of rows to select.

**Returns** **obj\_head** : type of caller

The first  $n$  rows of the caller object.

**See also:**

[`pandas.DataFrame.tail`](#) Returns the last  $n$  rows.

## Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0 alligator
1      bee
2   falcon
3     lion
4   monkey
5   parrot
6    shark
7    whale
8    zebra
```

Viewing the first 5 lines

```
>>> df.head()
   animal
0 alligator
1      bee
2   falcon
3     lion
4   monkey
```

Viewing the first  $n$  lines (three in this case)

```
>>> df.head(3)
   animal
0 alligator
1      bee
2   falcon
```

## pandas.DataFrame.hist

`DataFrame.hist` (*column=None, by=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, figsize=None, layout=None, bins=10, \*\*kws*)

Make a histogram of the DataFrame's.

A **histogram** is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

**Parameters** **data** : DataFrame

The pandas object holding the data.

**column** : string or sequence

If passed, will be used to limit data to a subset of columns.

**by** : object, optional

If passed, then used to form histograms for separate groups.

**grid** : boolean, default True

Whether to show axis grid lines.

**xlabelsize** : int, default None

If specified changes the x-axis label size.

**xrot** : float, default None

Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

**ylabelsize** : int, default None

If specified changes the y-axis label size.

**yrot** : float, default None

Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

**ax** : Matplotlib axes object, default None

The axes to plot the histogram on.

**sharex** : boolean, default True if ax is None else False

In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

**sharey** : boolean, default False

In case subplots=True, share y axis and set some y axis labels to invisible.

**figsize** : tuple

The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

**layout** : tuple, optional

Tuple of (rows, columns) for the layout of the histograms.

**bins** : integer or sequence, default 10

Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

**\*\*kwds**

All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

### Returns

**axes** [matplotlib.AxesSubplot or numpy.ndarray of them]

### See also:

**matplotlib.pyplot.hist** Plot a histogram using matplotlib.

### Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
...     }, index= ['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```

## pandas.DataFrame.idxmax

`DataFrame.idxmax` (*axis=0, skipna=True*)

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

### Returns

**idxmax** [Series]

### Raises ValueError

- If the row/column is empty

### See also:

`Series.idxmax`

### Notes

This method is the DataFrame version of `ndarray.argmax`.

## pandas.DataFrame.idxmin

`DataFrame.idxmin` (*axis=0, skipna=True*)

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

### Returns

**idxmin** [Series]

### Raises ValueError

- If the row/column is empty

### See also:

*Series.idxmin*

### Notes

This method is the DataFrame version of `ndarray.argmax`.

## pandas.DataFrame.infer\_objects

`DataFrame.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

### Returns

**converted** [same type as input object]

### See also:

*pandas.to\_datetime* Convert argument to datetime.

*pandas.to\_timedelta* Convert argument to timedelta.

*pandas.to\_numeric* Convert argument to numeric typeR

### Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A      int64
dtype: object
```

## pandas.DataFrame.info

`DataFrame.info(verbose=None, buf=None, max_cols=None, memory_usage=None, null_counts=None)`

Print a concise summary of a DataFrame.

This method prints information about a DataFrame including the index dtype and column dtypes, non-null values and memory usage.

**Parameters** `verbose` : bool, optional

Whether to print the full summary. By default, the setting in `pandas.options.display.max_info_columns` is followed.

**buf** : writable buffer, defaults to `sys.stdout`

Where to send the output. By default, the output is printed to `sys.stdout`. Pass a writable buffer if you need to further process the output.

**max\_cols** : int, optional

When to switch from the verbose to the truncated output. If the DataFrame has more than `max_cols` columns, the truncated output is used. By default, the setting in `pandas.options.display.max_info_columns` is used.

**memory\_usage** : bool, str, optional

Specifies whether total memory usage of the DataFrame elements (including the index) should be displayed. By default, this follows the `pandas.options.display.memory_usage` setting.

True always show memory usage. False never shows memory usage. A value of 'deep' is equivalent to "True with deep introspection". Memory usage is shown in human-readable units (base-2 representation). Without deep introspection a memory estimation is made based in column dtype and number of rows assuming values consume the same memory amount for corresponding dtypes. With deep memory introspection, a real memory usage calculation is performed at the cost of computational resources.

**null\_counts** : bool, optional

Whether to show the non-null counts. By default, this is shown only if the frame is smaller than `pandas.options.display.max_info_rows` and `pandas.options.display.max_info_columns`. A value of True always shows the counts, and False never shows the counts.

**Returns** None

This method prints a summary of a DataFrame and returns None.

See also:

[`DataFrame.describe`](#) Generate descriptive statistics of DataFrame columns.

[`DataFrame.memory\_usage`](#) Memory usage of DataFrame columns.

## Examples

```
>>> int_values = [1, 2, 3, 4, 5]
>>> text_values = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
>>> float_values = [0.0, 0.25, 0.5, 0.75, 1.0]
>>> df = pd.DataFrame({"int_col": int_values, "text_col": text_values,
...                     "float_col": float_values})
>>> df
   int_col text_col  float_col
0         1   alpha         0.00
1         2   beta         0.25
2         3  gamma         0.50
3         4  delta         0.75
4         5 epsilon         1.00
```

Prints information of all columns:

```
>>> df.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
int_col      5 non-null int64
text_col     5 non-null object
float_col    5 non-null float64
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Prints a summary of columns count and its dtypes but not per column information:

```
>>> df.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Columns: 3 entries, int_col to float_col
dtypes: float64(1), int64(1), object(1)
memory usage: 200.0+ bytes
```

Pipe output of DataFrame.info to buffer instead of sys.stdout, get buffer content and writes to a text file:

```
>>> import io
>>> buffer = io.StringIO()
>>> df.info(buf=buffer)
>>> s = buffer.getvalue()
>>> with open("df_info.txt", "w", encoding="utf-8") as f:
...     f.write(s)
260
```

The *memory\_usage* parameter allows deep introspection mode, specially useful for big DataFrames and fine-tune memory optimization:

```
>>> random_strings_array = np.random.choice(['a', 'b', 'c'], 10 ** 6)
>>> df = pd.DataFrame({
...     'column_1': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_2': np.random.choice(['a', 'b', 'c'], 10 ** 6),
...     'column_3': np.random.choice(['a', 'b', 'c'], 10 ** 6)
... })
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
```

(continues on next page)



(continued from previous page)

```

RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 22.9+ MB

```

```

>>> df.info(memory_usage='deep')
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 3 columns):
column_1    1000000 non-null object
column_2    1000000 non-null object
column_3    1000000 non-null object
dtypes: object(3)
memory usage: 188.8 MB

```

### pandas.DataFrame.insert

`DataFrame.insert` (*loc*, *column*, *value*, *allow\_duplicates=False*)

Insert column into DataFrame at specified location.

Raises a `ValueError` if *column* is already contained in the DataFrame, unless *allow\_duplicates* is set to `True`.

**Parameters** *loc* : int

Insertion index. Must verify  $0 \leq \text{loc} \leq \text{len}(\text{columns})$

**column** : string, number, or hashable object

label of the inserted column

**value** [int, Series, or array-like]

**allow\_duplicates** [bool, optional]

### pandas.DataFrame.interpolate

`DataFrame.interpolate` (*method='linear'*, *axis=0*, *limit=None*, *inplace=False*,  
*limit\_direction='forward'*, *limit\_area=None*, *downcast=None*,  
*\*\*kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

**Parameters** *method* : {'linear', 'time', 'index', 'values', 'nearest', 'zero',

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline',  
'piecewise\_polynomial', 'from\_derivatives', 'pchip', 'akima' }

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default

- ‘time’: interpolation works on daily and higher resolution data to interpolate given length of interval
- ‘index’, ‘values’: use the actual numerical values of the index
- ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘polynomial’ is passed to `scipy.interpolate.interpld`. Both ‘polynomial’ and ‘spline’ require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the actual numerical values of the index.
- ‘krogh’, ‘piecewise\_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the [scipy documentation](#) and [tutorial documentation](#)
- ‘from\_derivatives’ refers to `BPoly.from_derivatives` which replaces ‘piecewise\_polynomial’ interpolation method in scipy 0.18

New in version 0.18.1: Added support for the ‘akima’ method Added interpolate method ‘from\_derivatives’ which replaces ‘piecewise\_polynomial’ in scipy 0.18; backwards-compatible with scipy < 0.18

**axis** : {0, 1}, default 0

- 0: fill column-by-column
- 1: fill row-by-row

**limit** : int, default None.

Maximum number of consecutive NaNs to fill. Must be greater than 0.

**limit\_direction** [{‘forward’, ‘backward’, ‘both’}, default ‘forward’]

**limit\_area** : {‘inside’, ‘outside’}, default None

- None: (default) no fill restriction
- ‘inside’ Only fill NaNs surrounded by valid values (interpolate).
- ‘outside’ Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

**inplace** : bool, default False

Update the NDFrame in place if possible.

**downcast** : optional, ‘infer’ or None, defaults to None

Downcast dtypes if possible.

**kwargs** [keyword arguments to pass on to the interpolating function.]

## Returns

Series or DataFrame of same shape interpolated at the NaNs

See also:

[reindex](#), [replace](#), [fillna](#)

## Examples

### Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

## pandas.DataFrame.isin

`DataFrame.isin(values)`

Return boolean DataFrame showing whether each element in the DataFrame is contained in values.

**Parameters** `values` : iterable, Series, DataFrame or dictionary

The result will only be true at a location if all the labels match. If *values* is a Series, that's the index. If *values* is a dictionary, the keys must be the column names, which must match. If *values* is a DataFrame, then both the index and column labels must match.

**Returns**

**DataFrame of booleans**

## Examples

When values is a list:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> df.isin([1, 3, 12, 'a'])
   A    B
0  True  True
1 False False
2  True False
```

When values is a dict:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 4, 7]})
>>> df.isin({'A': [1, 3], 'B': [4, 7, 12]})
   A    B
0  True False # Note that B didn't match the 1 here.
1 False  True
2  True  True
```

When values is a Series or DataFrame:

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': ['a', 'b', 'f']})
>>> other = DataFrame({'A': [1, 3, 3, 2], 'B': ['e', 'f', 'f', 'e']})
>>> df.isin(other)
   A    B
0  True False
```

(continues on next page)

(continued from previous page)

```

1 False False # Column A in `other` has a 3, but not at index 1.
2  True  True

```

**pandas.DataFrame.isna**`DataFrame.isna()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns DataFrame**

Mask of bool values for each element in `DataFrame` that indicates whether an element is not an NA value.

**See also:**`DataFrame.isnull` alias of `isna``DataFrame.notna` boolean inverse of `isna``DataFrame.dropna` omit axes labels with missing values`isna` top-level `isna`**Examples**Show which entries in a `DataFrame` are NA.

```

>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born   name      toy
0  5.0      NaT  Alfred     None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker

```

```

>>> df.isna()
   age  born  name  toy
0 False  True False  True
1 False False False False
2  True False False False

```

Show which entries in a `Series` are NA.

```

>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0

```

(continues on next page)

(continued from previous page)

```
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

**pandas.DataFrame.isnull**`DataFrame.isnull()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns DataFrame**

Mask of bool values for each element in `DataFrame` that indicates whether an element is not an NA value.

**See also:**`DataFrame.isnull` alias of `isna``DataFrame.notna` boolean inverse of `isna``DataFrame.dropna` omit axes labels with missing values`isna` top-level `isna`**Examples**Show which entries in a `DataFrame` are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born   name      toy
0  5.0      NaT  Alfred     None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25     Joker     Joker
```

```
>>> df.isna()
   age  born  name  toy
0  False  True  False  True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2     NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

### **pandas.DataFrame.items**

`DataFrame.items()`

Iterator over (column name, Series) pairs.

**See also:**

*[iterrows](#)* Iterate over DataFrame rows as (index, Series) pairs.

*[itertuples](#)* Iterate over DataFrame rows as namedtuples of the values.

### **pandas.DataFrame.iteritems**

`DataFrame.iteritems()`

Iterator over (column name, Series) pairs.

**See also:**

*[iterrows](#)* Iterate over DataFrame rows as (index, Series) pairs.

*[itertuples](#)* Iterate over DataFrame rows as namedtuples of the values.

### **pandas.DataFrame.iterrows**

`DataFrame.iterrows()`

Iterate over DataFrame rows as (index, Series) pairs.

**Returns it :** generator

A generator that iterates over the rows of the frame.

**See also:**

*[itertuples](#)* Iterate over DataFrame rows as namedtuples of the values.

*[iteritems](#)* Iterate over (column name, Series) pairs.

## Notes

1. Because `iterrows` returns a Series for each row, it does **not** preserve dtypes across the rows (dtypes are preserved across columns for DataFrames). For example,

```
>>> df = pd.DataFrame([[1, 1.5]], columns=['int', 'float'])
>>> row = next(df.iterrows())[1]
>>> row
int      1.0
float    1.5
Name: 0, dtype: float64
>>> print(row['int'].dtype)
float64
>>> print(df['int'].dtype)
int64
```

To preserve dtypes while iterating over the rows, it is better to use `itertuples()` which returns namedtuples of the values and which is generally faster than `iterrows`.

2. You should **never modify** something you are iterating over. This is not guaranteed to work in all cases. Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

## pandas.DataFrame.itertuples

`DataFrame.itertuples` (*index=True, name='Pandas'*)

Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.

**Parameters** `index` : boolean, default True

If True, return the index as the first element of the tuple.

`name` : string, default "Pandas"

The name of the returned namedtuples or None to return regular tuples.

**See also:**

`iterrows` Iterate over DataFrame rows as (index, Series) pairs.

`iteritems` Iterate over (column name, Series) pairs.

## Notes

The column names will be renamed to positional names if they are invalid Python identifiers, repeated, or start with an underscore. With a large number of columns (>255), regular tuples are returned.

## Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [0.1, 0.2]},
                      index=['a', 'b'])
>>> df
   col1  col2
a      1   0.1
```

(continues on next page)

(continued from previous page)

```

b      2      0.2
>>> for row in df.itertuples():
...     print(row)
...
Pandas(Index='a', col1=1, col2=0.10000000000000001)
Pandas(Index='b', col1=2, col2=0.20000000000000001)

```

## pandas.DataFrame.join

`DataFrame.join(other, on=None, how='left', lsuffix="", rsuffix="", sort=False)`

Join columns with other DataFrame either on index or on a key column. Efficiently Join multiple DataFrame objects by index at once by passing a list.

**Parameters** **other** : DataFrame, Series with name field set, or list of DataFrame

Index should be similar to one of the columns in this one. If a Series is passed, its name attribute must be set, and that will be used as the column name in the resulting joined DataFrame

**on** : name, tuple/list of names, or array-like

Column or index level name(s) in the caller to join on the index in *other*, otherwise joins index-on-index. If multiple values given, the *other* DataFrame must have a MultiIndex. Can pass an array as the join key if it is not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

**how** : { 'left', 'right', 'outer', 'inner' }, default: 'left'

How to handle the operation of the two objects.

- left: use calling frame's index (or column if on is specified)
- right: use other frame's index
- outer: form union of calling frame's index (or column if on is specified) with other frame's index, and sort it lexicographically
- inner: form intersection of calling frame's index (or column if on is specified) with other frame's index, preserving the order of the calling's one

**lsuffix** : string

Suffix to use from left frame's overlapping columns

**rsuffix** : string

Suffix to use from right frame's overlapping columns

**sort** : boolean, default False

Order result DataFrame lexicographically by the join key. If False, the order of the join key depends on the join type (how keyword)

### Returns

**joined** [DataFrame]

See also:

[`DataFrame.merge`](#) For column(s)-on-columns(s) operations



## Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

Support for specifying index levels as the *on* parameter was added in version 0.23.0

## Examples

```
>>> caller = pd.DataFrame({'key': ['K0', 'K1', 'K2', 'K3', 'K4', 'K5'],
...                        'A': ['A0', 'A1', 'A2', 'A3', 'A4', 'A5']})
```

```
>>> caller
   A key
0  A0  K0
1  A1  K1
2  A2  K2
3  A3  K3
4  A4  K4
5  A5  K5
```

```
>>> other = pd.DataFrame({'key': ['K0', 'K1', 'K2'],
...                       'B': ['B0', 'B1', 'B2']})
```

```
>>> other
   B key
0  B0  K0
1  B1  K1
2  B2  K2
```

Join DataFrames using their indexes.

```
>>> caller.join(other, lsuffix='_caller', rsuffix='_other')
```

```
>>>
   A key_caller  B key_other
0  A0          K0  B0          K0
1  A1          K1  B1          K1
2  A2          K2  B2          K2
3  A3          K3  NaN         NaN
4  A4          K4  NaN         NaN
5  A5          K5  NaN         NaN
```

If we want to join using the key columns, we need to set key to be the index in both caller and other. The joined DataFrame will have key as its index.

```
>>> caller.set_index('key').join(other.set_index('key'))
```

```
>>>
   A    B
key
K0  A0  B0
K1  A1  B1
K2  A2  B2
K3  A3  NaN
K4  A4  NaN
K5  A5  NaN
```

Another option to join using the key columns is to use the `on` parameter. `DataFrame.join` always uses other's index but we can use any column in the caller. This method preserves the original caller's index in the result.

```
>>> caller.join(other.set_index('key'), on='key')
```

```
>>>
   A key  B
0  A0  K0  B0
1  A1  K1  B1
2  A2  K2  B2
3  A3  K3  NaN
4  A4  K4  NaN
5  A5  K5  NaN
```

### **pandas.DataFrame.keys**

`DataFrame.keys()`

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and `major_axis` for Panel.

### **pandas.DataFrame.kurt**

`DataFrame.kurt` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

#### **Parameters**

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### **Returns**

**kurt** [Series or DataFrame (if level specified)]

### **pandas.DataFrame.kurtosis**

`DataFrame.kurtosis` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

#### **Parameters**

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**kurt** [Series or DataFrame (if level specified)]

## pandas.DataFrame.last

`DataFrame.last` (*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

### Parameters

**offset** [string, DateOffset, dateutil.relativedelta]

### Returns

**subset** [type of caller]

### Raises TypeError

If the index is not a *DatetimeIndex*

### See also:

**first** Select initial periods of time series based on a date offset

**at\_time** Select values at a particular time of the day

**between\_time** Select values between particular times of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09	1
2018-04-11	2
2018-04-13	3
2018-04-15	4

Get the rows for the last 3 days:

```
>>> ts.last('3D')
```

	A
2018-04-13	3
2018-04-15	4

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

### **pandas.DataFrame.last\_valid\_index**

`DataFrame.last_valid_index()`  
Return index for last non-NA/null value.

#### **Returns**

**scalar** [type of index]

#### **Notes**

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

### **pandas.DataFrame.le**

`DataFrame.le(other, axis='columns', level=None)`  
Wrapper for flexible comparison methods `le`

### **pandas.DataFrame.lookup**

`DataFrame.lookup(row_labels, col_labels)`  
Label-based “fancy indexing” function for DataFrame. Given equal-length arrays of row and column labels, return an array of the values corresponding to each (row, col) pair.

**Parameters** `row_labels` : sequence

The row labels to use for lookup

`col_labels` : sequence

The column labels to use for lookup

#### **Notes**

Akin to:

```
result = []
for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))
```

#### **Examples**

**values** [ndarray] The found values

**pandas.DataFrame.lt**`DataFrame.lt` (*other*, *axis*='columns', *level*=None)Wrapper for flexible comparison methods `lt`**pandas.DataFrame.mad**`DataFrame.mad` (*axis*=None, *skipna*=None, *level*=None)

Return the mean absolute deviation of the values for the requested axis

**Parameters****axis** [{index (0), columns (1)}]**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns****mad** [Series or DataFrame (if level specified)]**pandas.DataFrame.mask**`DataFrame.mask` (*cond*, *other*=nan, *inplace*=False, *axis*=None, *level*=None, *errors*='raise',  
*try\_cast*=False, *raise\_on\_error*=None)Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.**Parameters** **cond** : boolean NDFrame, array-like, or callableWhere *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).New in version 0.18.1: A callable can be used as *cond*.**other** : scalar, NDFrame, or callableEntries where *cond* is True are replaced with corresponding value from *other*. If *other* is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).New in version 0.18.1: A callable can be used as *other*.**inplace** : boolean, default False

Whether to perform the operation in place on the data

**axis** [alignment axis if needed, default None]

**level** [alignment level if needed, default None]

**errors** : str, {'raise', 'ignore'}, default 'raise'

- **raise** : allow exceptions to be raised
- **ignore** : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

### Returns

**wh** [same type as caller]

### See also:

[`DataFrame.where\(\)`](#)

### Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for [`DataFrame.where\(\)`](#) differs from [`numpy.where\(\)`](#). Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in [indexing](#).

### Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1     NaN
2     NaN
3     NaN
4     NaN
```

```

>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0

>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True

```

### pandas.DataFrame.max

DataFrame.**max** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

**This method returns the maximum of the values in the object.** If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

#### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**max** [Series or DataFrame (if level specified)]

## pandas.DataFrame.mean

DataFrame.**mean** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return the mean of the values for the requested axis

### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**mean** [Series or DataFrame (if level specified)]

## pandas.DataFrame.median

DataFrame.**median** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return the median of the values for the requested axis

### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**median** [Series or DataFrame (if level specified)]

## pandas.DataFrame.melt

DataFrame.**melt** (*id\_vars=None, value\_vars=None, var\_name=None, value\_name='value', col\_level=None*)

“Unpivots” a DataFrame from wide format to long format, optionally leaving identifier variables set.



This function is useful to massage a DataFrame into a format where one or more columns are identifier variables (*id\_vars*), while all other columns, considered measured variables (*value\_vars*), are “unpivoted” to the row axis, leaving just two non-identifier columns, ‘variable’ and ‘value’.

New in version 0.20.0.

### Parameters

- frame** [DataFrame]  
**id\_vars** : tuple, list, or ndarray, optional  
 Column(s) to use as identifier variables.
- value\_vars** : tuple, list, or ndarray, optional  
 Column(s) to unpivot. If not specified, uses all columns that are not set as *id\_vars*.
- var\_name** : scalar  
 Name to use for the ‘variable’ column. If None it uses `frame.columns.name` or ‘variable’.
- value\_name** : scalar, default ‘value’  
 Name to use for the ‘value’ column.
- col\_level** : int or string, optional  
 If columns are a MultiIndex then use this level to melt.

### See also:

[`melt`](#), [`pivot\_table`](#), [`DataFrame.pivot`](#)

### Examples

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': {0: 'a', 1: 'b', 2: 'c'},
...                   'B': {0: 1, 1: 3, 2: 5},
...                   'C': {0: 2, 1: 4, 2: 6}})
>>> df
   A  B  C
0  a  1  2
1  b  3  4
2  c  5  6
```

```
>>> df.melt(id_vars=['A'], value_vars=['B'])
   A variable  value
0  a         B      1
1  b         B      3
2  c         B      5
```

```
>>> df.melt(id_vars=['A'], value_vars=['B', 'C'])
   A variable  value
0  a         B      1
1  b         B      3
2  c         B      5
3  a         C      2
4  b         C      4
5  c         C      6
```

The names of ‘variable’ and ‘value’ columns can be customized:

```
>>> df.melt(id_vars=['A'], value_vars=['B'],
...         var_name='myVarname', value_name='myValname')
   A myVarname myValname
0  a         B         1
1  b         B         3
2  c         B         5
```

If you have multi-index columns:

```
>>> df.columns = [list('ABC'), list('DEF')]
>>> df
   A B C
   D E F
0  a 1 2
1  b 3 4
2  c 5 6
```

```
>>> df.melt(col_level=0, id_vars=['A'], value_vars=['B'])
   A variable  value
0  a         B      1
1  b         B      3
2  c         B      5
```

```
>>> df.melt(id_vars=[('A', 'D')], value_vars=[('B', 'E')])
   (A, D) variable_0 variable_1  value
0      a           B           E      1
1      b           B           E      3
2      c           B           E      5
```

## pandas.DataFrame.memory\_usage

`DataFrame.memory_usage(index=True, deep=False)`

Return the memory usage of each column in bytes.

The memory usage can optionally include the contribution of the index and elements of *object* dtype.

This value is displayed in *DataFrame.info* by default. This can be suppressed by setting `pandas.options.display.memory_usage` to False.

**Parameters** `index` : bool, default True

Specifies whether to include the memory usage of the DataFrame’s index in returned Series. If `index=True` the memory usage of the index the first item in the output.

**deep** : bool, default False

If True, introspect the data deeply by interrogating *object* dtypes for system-level memory consumption, and include it in the returned values.

**Returns** `sizes` : Series

A Series whose index is the original column names and whose values is the memory usage of each column in bytes.

**See also:**

**`numpy.ndarray.nbytes`** Total bytes consumed by the elements of an ndarray.

**`Series.memory_usage`** Bytes consumed by a Series.

**`pandas.Categorical`** Memory-efficient array for string values with many repeated values.

**`DataFrame.info`** Concise summary of a DataFrame.

## Examples

```
>>> dtypes = ['int64', 'float64', 'complex128', 'object', 'bool']
>>> data = dict([(t, np.ones(shape=5000).astype(t))
...              for t in dtypes])
>>> df = pd.DataFrame(data)
>>> df.head()
   int64  float64  complex128  object  bool
0      1      1.0      (1+0j)      1  True
1      1      1.0      (1+0j)      1  True
2      1      1.0      (1+0j)      1  True
3      1      1.0      (1+0j)      1  True
4      1      1.0      (1+0j)      1  True
```

```
>>> df.memory_usage()
Index          80
int64         40000
float64        40000
complex128     80000
object         40000
bool           5000
dtype: int64
```

```
>>> df.memory_usage(index=False)
int64         40000
float64        40000
complex128     80000
object         40000
bool           5000
dtype: int64
```

The memory footprint of *object* dtype columns is ignored by default:

```
>>> df.memory_usage(deep=True)
Index          80
int64         40000
float64        40000
complex128     80000
object        160000
bool           5000
dtype: int64
```

Use a Categorical for efficient storage of an object-dtype column with many repeated values.

```
>>> df['object'].astype('category').memory_usage(deep=True)
5168
```

## pandas.DataFrame.merge

```
DataFrame.merge(right, how='inner', on=None, left_on=None, right_on=None, left_index=False,
                 right_index=False, sort=False, suffixes=('_x', '_y'), copy=True, indica-
                 tor=False, validate=None)
```

Merge DataFrame objects by performing a database-style join operation by columns or indexes.

If joining columns on columns, the DataFrame indexes *will be ignored*. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on.

### Parameters

**right** [DataFrame]

**how** : { 'left', 'right', 'outer', 'inner' }, default 'inner'

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order
- outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys

**on** : label or list

Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

**left\_on** : label or list, or array-like

Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

**right\_on** : label or list, or array-like

Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

**left\_index** : boolean, default False

Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels

**right\_index** : boolean, default False

Use the index from the right DataFrame as the join key. Same caveats as left\_index

**sort** : boolean, default False

Sort the join keys lexicographically in the result DataFrame. If False, the order of the join keys depends on the join type (how keyword)

**suffixes** : 2-length sequence (tuple, list, ...)

Suffix to apply to overlapping column names in the left and right side, respectively

**copy** : boolean, default True

If False, do not copy data unnecessarily

**indicator** : boolean or string, default False

If True, adds a column to output DataFrame called “\_merge” with information on the source of each row. If string, column with information on source of each row will be added to output DataFrame, and column will be named value of string. Information column is Categorical-type and takes on a value of “left\_only” for observations whose merge key only appears in ‘left’ DataFrame, “right\_only” for observations whose merge key only appears in ‘right’ DataFrame, and “both” if the observation’s merge key is found in both.

**validate** : string, default None

If specified, checks if merge is of specified type.

- “one\_to\_one” or “1:1”: check if merge keys are unique in both left and right datasets.
- “one\_to\_many” or “1:m”: check if merge keys are unique in left dataset.
- “many\_to\_one” or “m:1”: check if merge keys are unique in right dataset.
- “many\_to\_many” or “m:m”: allowed, but does not result in checks.

New in version 0.21.0.

**Returns** **merged** : DataFrame

The output type will be the same as ‘left’, if it is a subclass of DataFrame.

**See also:**

[\*merge\\_ordered\*](#), [\*merge\\_asof\*](#), [\*DataFrame.join\*](#)

## Notes

Support for specifying index levels as the *on*, *left\_on*, and *right\_on* parameters was added in version 0.23.0

## Examples

```
>>> A          >>> B
   lkey value   rkey value
0   foo    1    0   foo    5
1   bar    2    1   bar    6
2   baz    3    2   qux    7
3   foo    4    3   bar    8
```

```
>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0   foo      1    foo      5
1   foo      4    foo      5
2   bar      2    bar      6
3   bar      2    bar      8
4   baz      3   NaN     NaN
5   NaN     NaN   qux      7
```

### pandas.DataFrame.min

`DataFrame.min` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

**This method returns the minimum of the values in the object.** If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

#### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**min** [Series or DataFrame (if level specified)]

### pandas.DataFrame.mod

`DataFrame.mod` (*other, axis='columns', level=None, fill\_value=None*)

Modulo of dataframe and other, element-wise (binary operator *mod*).

Equivalent to `dataframe % other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

**See also:**

`DataFrame.rmod`

## Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.mode

`DataFrame.mode(axis=0, numeric_only=False)`

Gets the mode(s) of each element along the axis selected. Adds a row for each mode per label, fills in gaps with nan.

Note that there could be multiple values returned for the selected axis (when more than one item share the maximum frequency), which is the reason why a dataframe is returned. If you want to impute missing values with the mode in a dataframe `df`, you can just do this: `df.fillna(df.mode().iloc[0])`

**Parameters** `axis`: {0 or 'index', 1 or 'columns'}, default 0

- 0 or 'index': get mode of each column
- 1 or 'columns': get mode of each row

**numeric\_only**: boolean, default False

if True, only apply to numeric columns

#### Returns

**modes** [DataFrame (sorted)]

## Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 1, 2, 1, 2, 3]})
>>> df.mode()
   A
0  1
1  2
```

### pandas.DataFrame.mul

`DataFrame.mul(other, axis='columns', level=None, fill_value=None)`

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis**: {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

#### See also:

[`DataFrame.rmul`](#)

#### Notes

Mismatched indices will be unioned together

#### Examples

None

### **pandas.DataFrame.multiply**

`DataFrame.multiply` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

#### See also:

[`DataFrame.rmul`](#)



## Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.ne

`DataFrame.ne` (*other*, *axis*='columns', *level*=None)  
Wrapper for flexible comparison methods `ne`

### pandas.DataFrame.nlargest

`DataFrame.nlargest` (*n*, *columns*, *keep*='first')

Return the first *n* rows ordered by *columns* in descending order.

Return the first *n* rows with the largest values in *columns*, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to `df.sort_values(columns, ascending=False).head(n)`, but more performant.

**Parameters** *n* : int

Number of rows to return.

**columns** : label or list of labels

Column label(s) to order by.

**keep** : {'first', 'last'}, default 'first'

Where there are duplicate values:

- *first* : prioritize the first occurrence(s)
- *last* : prioritize the last occurrence(s)

**Returns** `DataFrame`

The first *n* rows ordered by the given columns in descending order.

**See also:**

`DataFrame.nsmallest` Return the first *n* rows ordered by *columns* in ascending order.

`DataFrame.sort_values` Sort `DataFrame` by the values

`DataFrame.head` Return the first *n* rows without re-ordering.

## Notes

This function cannot be used with all column types. For example, when specifying columns with *object* or *category* dtypes, `TypeError` is raised.

## Examples

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 10, -1],
...                     'b': list('abdce'),
...                     'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df
```

	a	b	c
0	1	a	1.0
1	10	b	2.0
2	8	d	NaN
3	10	c	3.0
4	-1	e	4.0

In the following example, we will use `nlargest` to select the three rows having the largest values in column “a”.

```
>>> df.nlargest(3, 'a')
```

	a	b	c
1	10	b	2.0
3	10	c	3.0
2	8	d	NaN

When using `keep='last'`, ties are resolved in reverse order:

```
>>> df.nlargest(3, 'a', keep='last')
```

	a	b	c
3	10	c	3.0
1	10	b	2.0
2	8	d	NaN

To order by the largest values in column “a” and then “c”, we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['a', 'c'])
```

	a	b	c
3	10	c	3.0
1	10	b	2.0
2	8	d	NaN

Attempting to use `nlargest` on non-numeric dtypes will raise a `TypeError`:

```
>>> df.nlargest(3, 'b')
Traceback (most recent call last):
TypeError: Column 'b' has dtype object, cannot use method 'nlargest'
```

## pandas.DataFrame.notna

`DataFrame.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to `False` values.

**Returns DataFrame**

Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

See also:

`DataFrame.notnull` alias of `notna`

`DataFrame.isna` boolean inverse of `notna`

`DataFrame.dropna` omit axes labels with missing values

`notna` top-level `notna`

## Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born   name    toy
0  5.0      NaT  Alfred   None
1  6.0 1939-05-27  Batman Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notna()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0    True
1    True
2   False
dtype: bool
```

## pandas.DataFrame.notnull

`DataFrame.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set

`pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to `False` values.

### Returns DataFrame

Mask of bool values for each element in `DataFrame` that indicates whether an element is not an NA value.

### See also:

`DataFrame.notnull` alias of `notna`

`DataFrame.isna` boolean inverse of `notna`

`DataFrame.dropna` omit axes labels with missing values

`notna` top-level `notna`

### Examples

Show which entries in a `DataFrame` are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born   name    toy
0  5.0      NaT  Alfred   None
1  6.0 1939-05-27  Batman Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notna()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a `Series` are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0    True
1    True
2   False
dtype: bool
```

**pandas.DataFrame.nsmallest**

`DataFrame.nsmallest` (*n*, *columns*, *keep*='first')

Get the rows of a DataFrame sorted by the *n* smallest values of *columns*.

**Parameters** *n* : int

Number of items to retrieve

**columns** : list or str

Column name or names to order by

**keep** : {'first', 'last'}, default 'first'

Where there are duplicate values: - *first* : take the first occurrence. - *last* : take the last occurrence.

**Returns**

**DataFrame**

**Examples**

```
>>> df = pd.DataFrame({'a': [1, 10, 8, 11, -1],
...                     'b': list('abdce'),
...                     'c': [1.0, 2.0, np.nan, 3.0, 4.0]})
>>> df.nsmallest(3, 'a')
   a  b  c
4 -1  e  4
0  1  a  1
2  8  d NaN
```

**pandas.DataFrame.nunique**

`DataFrame.nunique` (*axis*=0, *dropna*=True)

Return Series with number of distinct observations over requested axis.

New in version 0.20.0.

**Parameters**

**axis** [{0 or 'index', 1 or 'columns'}, default 0]

**dropna** : boolean, default True

Don't include NaN in the counts.

**Returns**

**nunique** [Series]

**Examples**

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [1, 1, 1]})
>>> df.nunique()
A    3
B    1
```

```
>>> df.nunique(axis=1)
0    1
1    2
2    2
```

## pandas.DataFrame.pct\_change

`DataFrame.pct_change` (*periods=1, fill\_method='pad', limit=None, freq=None, \*\*kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

**Parameters** *periods* : int, default 1

Periods to shift for forming percent change.

*fill\_method* : str, default 'pad'

How to handle NAs before computing percent changes.

*limit* : int, default None

The number of consecutive NAs to fill before stopping.

*freq* : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay()).

**\*\*kwargs**

Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

**Returns** *chg* : Series or DataFrame

The same type as the calling object.

**See also:**

[\*Series.diff\*](#) Compute the difference of two elements in a Series.

[\*DataFrame.diff\*](#) Compute the difference of two elements in a DataFrame.

[\*Series.shift\*](#) Shift the index by some number of periods.

[\*DataFrame.shift\*](#) Shift the index by some number of periods.

## Examples

### Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0      NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0      NaN
1      NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2      NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0      NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

## DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
```

	FR	GR	IT
1980-01-01	4.0405	1.7246	804.74
1980-02-01	4.0963	1.7482	810.01
1980-03-01	4.3149	1.8519	860.13

```
>>> df.pct_change()
           FR          GR          IT
1980-01-01   NaN        NaN        NaN
1980-02-01  0.013810  0.013684  0.006549
1980-03-01  0.053365  0.059318  0.061876
```

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
```

(continues on next page)

(continued from previous page)

```
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
      2016      2015      2014
GOOG  1769950  1500923  1371819
APPL  30586265  40912316  41403351
```

```
>>> df.pct_change(axis='columns')
      2016      2015      2014
GOOG   NaN -0.151997 -0.086016
APPL   NaN  0.337604  0.012002
```

## pandas.DataFrame.pipe

`DataFrame.pipe(func, *args, **kwargs)`

Apply func(self, \*args, \*\*kwargs)

**Parameters** `func` : function

function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a (callable, `data_keyword`) tuple where `data_keyword` is a string indicating the keyword of callable that expects the NDFrame.

`args` : iterable, optional

positional arguments passed into `func`.

`kwargs` : mapping, optional

a dictionary of keyword arguments passed into `func`.

**Returns**

**object** [the return type of `func`.]

**See also:**

`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`

## Notes

Use `.pipe` when chaining together functions that expect Series, DataFrames or GroupBy objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe(f, arg2=b, arg3=c)
... )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:



```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe((f, 'arg2'), arg1=a, arg3=c)
...   )
```

## pandas.DataFrame.pivot

`DataFrame.pivot` (*index=None, columns=None, values=None*)

Return reshaped DataFrame organized by given index / column values.

Reshape data (produce a “pivot” table) based on column values. Uses unique values from specified *index* / *columns* to form axes of the resulting DataFrame. This function does not support data aggregation, multiple values will result in a MultiIndex in the columns. See the [User Guide](#) for more on reshaping.

**Parameters** *index* : string or object, optional

Column to use to make new frame’s index. If None, uses existing index.

**columns** : string or object

Column to use to make new frame’s columns.

**values** : string, object or a list of the previous, optional

Column(s) to use for populating new frame’s values. If not specified, all remaining columns will be used and the result will have hierarchically indexed columns.

Changed in version 0.23.0: Also accept list of column names.

**Returns** `DataFrame`

Returns reshaped DataFrame.

**Raises** `ValueError`:

When there are any *index*, *columns* combinations with multiple values.  
*DataFrame.pivot\_table* when you need to aggregate.

**See also:**

[`DataFrame.pivot\_table`](#) generalization of pivot that can handle duplicate values for one index/column pair.

[`DataFrame.unstack`](#) pivot based on the index values instead of a column.

## Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods.

## Examples

```
>>> df = pd.DataFrame({'foo': ['one', 'one', 'one', 'two', 'two',
...                             'two'],
...                    'bar': ['A', 'B', 'C', 'A', 'B', 'C'],
...                    'baz': [1, 2, 3, 4, 5, 6],
```

(continues on next page)

(continued from previous page)

```
...             'zoo': ['x', 'y', 'z', 'q', 'w', 't'])
>>> df
   foo bar baz zoo
0  one  A   1   x
1  one  B   2   y
2  one  C   3   z
3  two  A   4   q
4  two  B   5   w
5  two  C   6   t
```

```
>>> df.pivot(index='foo', columns='bar', values='baz')
bar A    B    C
foo
one 1    2    3
two 4    5    6
```

```
>>> df.pivot(index='foo', columns='bar')['baz']
bar A    B    C
foo
one 1    2    3
two 4    5    6
```

```
>>> df.pivot(index='foo', columns='bar', values=['baz', 'zoo'])
      baz      zoo
bar  A  B  C  A  B  C
foo
one   1  2  3   x  y  z
two   4  5  6   q  w  t
```

A `ValueError` is raised if there are any duplicates.

```
>>> df = pd.DataFrame({"foo": ['one', 'one', 'two', 'two'],
...                     "bar": ['A', 'A', 'B', 'C'],
...                     "baz": [1, 2, 3, 4]})
>>> df
   foo bar baz
0  one  A   1
1  one  A   2
2  two  B   3
3  two  C   4
```

Notice that the first two rows are the same for our *index* and *columns* arguments.

```
>>> df.pivot(index='foo', columns='bar', values='baz')
Traceback (most recent call last):
...
ValueError: Index contains duplicate entries, cannot reshape
```

## pandas.DataFrame.pivot\_table

`DataFrame.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')`

Create a spreadsheet-style pivot table as a `DataFrame`. The levels in the pivot table will be stored in `MultiIndex` objects (hierarchical indexes) on the index and columns of the result `DataFrame`

**Parameters****values** [column to aggregate, optional]**index** : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.

**columns** : column, Grouper, array, or list of the previous

If an array is passed, it must be the same length as the data. The list can contain any of the other types (except list). Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.

**aggfunc** : function, list of functions, dict, default numpy.mean

If list of functions passed, the resulting pivot table will have hierarchical columns whose top level are the function names (inferred from the function objects themselves) If dict is passed, the key is column to aggregate and value is function or list of functions

**fill\_value** : scalar, default None

Value to replace missing values with

**margins** : boolean, default False

Add all row / columns (e.g. for subtotal / grand totals)

**dropna** : boolean, default True

Do not include columns whose entries are all NaN

**margins\_name** : string, default 'All'

Name of the row / column that will contain the totals when margins is True.

**Returns****table** [DataFrame]**See also:***DataFrame.pivot* pivot without aggregation that can handle non-numeric data**Examples**

```
>>> df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
...                           "bar", "bar", "bar", "bar"],
...                    "B": ["one", "one", "one", "two", "two",
...                           "one", "one", "two", "two"],
...                    "C": ["small", "large", "large", "small",
...                           "small", "large", "small", "small",
...                           "large"],
...                    "D": [1, 2, 2, 3, 3, 4, 5, 6, 7]})
>>> df
   A  B  C  D
0  foo one small 1
1  foo one large 2
2  foo one large 2
```

(continues on next page)

(continued from previous page)

```

3  foo  two  small  3
4  foo  two  small  3
5  bar  one  large  4
6  bar  one  small  5
7  bar  two  small  6
8  bar  two  large  7

```

```

>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                      columns=['C'], aggfunc=np.sum)
>>> table
C      large  small
A  B
bar one    4.0    5.0
   two    7.0    6.0
foo one    4.0    1.0
   two    NaN    6.0

```

```

>>> table = pivot_table(df, values='D', index=['A', 'B'],
...                      columns=['C'], aggfunc=np.sum)
>>> table
C      large  small
A  B
bar one    4.0    5.0
   two    7.0    6.0
foo one    4.0    1.0
   two    NaN    6.0

```

```

>>> table = pivot_table(df, values=['D', 'E'], index=['A', 'C'],
...                      aggfunc={'D': np.mean,
...                                'E': [min, max, np.mean]})
>>> table
      D      E
      mean max median min
A  C
bar large  5.500000  16   14.5  13
   small  5.500000  15   14.5  14
foo large  2.000000  10    9.5   9
   small  2.333333  12   11.0   8

```

## pandas.DataFrame.plot

`DataFrame.plot` (*x=None*, *y=None*, *kind='line'*, *ax=None*, *subplots=False*, *sharex=None*, *sharey=False*, *layout=None*, *figsize=None*, *use\_index=True*, *title=None*, *grid=None*, *legend=True*, *style=None*, *logx=False*, *logy=False*, *loglog=False*, *xticks=None*, *yticks=None*, *xlim=None*, *ylim=None*, *rot=None*, *fontsize=None*, *colormap=None*, *table=False*, *yerr=None*, *xerr=None*, *secondary\_y=False*, *sort\_columns=False*, *\*\*kwds*)

Make plots of DataFrame using matplotlib / pylab.

*New in version 0.17.0:* Each plot kind has a corresponding method on the `DataFrame.plot` accessor: `df.plot(kind='line')` is equivalent to `df.plot.line()`.

### Parameters

**data** [DataFrame]

**x** [label or position, default None]

**y** : label, position or list of label, positions, default None

Allows plotting of one column versus another

**kind** : str

- 'line' : line plot (default)
- 'bar' : vertical bar plot
- 'barh' : horizontal bar plot
- 'hist' : histogram
- 'box' : boxplot
- 'kde' : Kernel Density Estimation plot
- 'density' : same as 'kde'
- 'area' : area plot
- 'pie' : pie plot
- 'scatter' : scatter plot
- 'hexbin' : hexbin plot

**ax** [matplotlib axes object, default None]

**subplots** : boolean, default False

Make separate subplots for each column

**sharex** : boolean, default True if ax is None else False

In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in; Be aware, that passing in both an ax and sharex=True will alter all x axis labels for all axis in a figure!

**sharey** : boolean, default False

In case subplots=True, share y axis and set some y axis labels to invisible

**layout** : tuple (optional)

(rows, columns) for the layout of subplots

**figsize** [a tuple (width, height) in inches]

**use\_index** : boolean, default True

Use index as ticks for x axis

**title** : string or list

Title to use for the plot. If a string is passed, print the string at the top of the figure. If a list is passed and *subplots* is True, print each item in the list above the corresponding subplot.

**grid** : boolean, default None (matlab style default)

Axis grid lines

**legend** : False/True/'reverse'

Place legend on axis subplots

**style** : list or dict

matplotlib line style per column

**logx** : boolean, default False

Use log scaling on x axis

**logy** : boolean, default False

Use log scaling on y axis

**loglog** : boolean, default False

Use log scaling on both x and y axes

**xticks** : sequence

Values to use for the xticks

**yticks** : sequence

Values to use for the yticks

**xlim** [2-tuple/list]

**ylim** [2-tuple/list]

**rot** : int, default None

Rotation for ticks (xticks for vertical, yticks for horizontal plots)

**fontsize** : int, default None

Font size for xticks and yticks

**colormap** : str or matplotlib colormap object, default None

Colormap to select colors from. If string, load colormap with that name from matplotlib.

**colorbar** : boolean, optional

If True, plot colorbar (only relevant for ‘scatter’ and ‘hexbin’ plots)

**position** : float

Specify relative alignments for bar plot layout. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)

**table** : boolean, Series or DataFrame, default False

If True, draw a table using the data in the DataFrame and the data will be transposed to meet matplotlib’s default layout. If a Series or DataFrame is passed, use passed data to draw a table.

**yerr** : DataFrame, Series, array-like, dict and str

See [Plotting with Error Bars](#) for detail.

**xerr** [same types as yerr.]

**stacked** : boolean, default False in line and

bar plots, and True in area plot. If True, create stacked plot.

**sort\_columns** : boolean, default False

Sort column names to determine plot ordering

**secondary\_y** : boolean or sequence, default False

Whether to plot on the secondary y-axis If a list/tuple, which columns to plot on secondary y-axis

**mark\_right** : boolean, default True

When using a secondary\_y axis, automatically mark the column labels with “(right)” in the legend

**\*\*kwargs** : keywords

Options to pass to matplotlib plotting method

### Returns

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

### Notes

- See matplotlib documentation online for more on this subject
- If *kind* = ‘bar’ or ‘barh’, you can specify relative alignments for bar plot layout by *position* keyword. From 0 (left/bottom-end) to 1 (right/top-end). Default is 0.5 (center)
- If *kind* = ‘scatter’ and the argument *c* is the name of a dataframe column, the values of that column are used to color each point.
- If *kind* = ‘hexbin’, you can control the size of the bins with the *gridsize* argument. By default, a histogram of the counts around each (*x*, *y*) point is computed. You can specify alternative aggregations by passing values to the *C* and *reduce\_C\_function* arguments. *C* specifies the value at each (*x*, *y*) point and *reduce\_C\_function* is a function of one argument that reduces all the values in a bin to a single number (e.g. *mean*, *max*, *sum*, *std*).

## pandas.DataFrame.pop

DataFrame.**pop** (*item*)

Return item and drop from frame. Raise KeyError if not found.

**Parameters** *item* : str

Column label to be popped

### Returns

**popped** [Series]

### Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                    ('parrot', 'bird', 24.0),
...                    ('lion', 'mammal', 80.5),
...                    ('monkey', 'mammal', np.nan)],
...                    columns=('name', 'class', 'max_speed'))
```

(continues on next page)

(continued from previous page)

```
>>> df
   name  class  max_speed
0  falcon   bird    389.0
1  parrot   bird     24.0
2   lion  mammal     80.5
3  monkey  mammal      NaN
```

```
>>> df.pop('class')
0    bird
1    bird
2  mammal
3  mammal
Name: class, dtype: object
```

```
>>> df
   name  max_speed
0  falcon    389.0
1  parrot    24.0
2   lion    80.5
3  monkey     NaN
```

**pandas.DataFrame.pow**

`DataFrame.pow` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Exponential power of dataframe and other, element-wise (binary operator *pow*).

Equivalent to `dataframe ** other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

**Parameters**

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

**Returns**

**result** [DataFrame]

**See also:**

[`DataFrame.rpow`](#)

**Notes**

Mismatched indices will be unioned together



## Examples

None

### pandas.DataFrame.prod

`DataFrame.prod(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the product of the values for the requested axis

#### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

#### Returns

**prod** [Series or DataFrame (if level specified)]

## Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

## pandas.DataFrame.product

`DataFrame.product` (*axis=None, skipna=None, level=None, numeric\_only=None, min\_count=0, \*\*kwargs*)

Return the product of the values for the requested axis

### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

### Returns

**prod** [Series or DataFrame (if level specified)]

## Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

**pandas.DataFrame.quantile**

`DataFrame.quantile` (*q=0.5, axis=0, numeric\_only=True, interpolation='linear'*)

Return values at the given quantile over requested axis, a la `numpy.percentile`.

**Parameters** *q* : float or array-like, default 0.5 (50% quantile)

0 <= *q* <= 1, the quantile(s) to compute

**axis** : {0, 1, 'index', 'columns'} (default 0)

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**numeric\_only** : boolean, default True

If False, the quantile of datetime and timedelta data will be computed as well

**interpolation** : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}

New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear:  $i + (j - i) * \text{fraction}$ , where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint:  $(i + j) / 2$ .

**Returns** **quantiles** : Series or DataFrame

- If *q* is an array, a DataFrame will be returned where the index is *q*, the columns are the columns of self, and the values are the quantiles.
- If *q* is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

**See also:**

`pandas.core.window.Rolling.quantile`

**Examples**

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                        columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
      a    b
0.1  1.3  3.7
0.5  2.5 55.0
```

Specifying *numeric\_only=False* will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                        'B': [pd.Timestamp('2010'),
                              pd.Timestamp('2011')],
                        'C': [pd.Timedelta('1 days'),
                              pd.Timedelta('2 days')]])
>>> df.quantile(0.5, numeric_only=False)
A          1.5
B    2010-07-02 12:00:00
C          1 days 12:00:00
Name: 0.5, dtype: object
```

## pandas.DataFrame.query

`DataFrame.query` (*expr*, *inplace=False*, *\*\*kwargs*)

Query the columns of a frame with a boolean expression.

**Parameters** *expr* : string

The query string to evaluate. You can refer to variables in the environment by prefixing them with an '@' character like @a + b.

**inplace** : bool

Whether the query should modify the data in place or return a modified copy

New in version 0.18.0.

**kwargs** : dict

See the documentation for `pandas.eval()` for complete details on the keyword arguments accepted by `DataFrame.query()`.

**Returns**

**q** [DataFrame]

**See also:**

`pandas.eval`, `DataFrame.eval`

## Notes

The result of the evaluation of this expression is first passed to `DataFrame.loc` and if that fails because of a multidimensional key (e.g., a DataFrame) then the result will be passed to `DataFrame.__getitem__()`.

This method uses the top-level `pandas.eval()` function to evaluate the passed query.

The `query()` method uses a slightly modified Python syntax by default. For example, the & and | (bitwise) operators have the precedence of their boolean cousins, `and` and `or`. This *is* syntactically valid Python, however the semantics are different.

You can change the semantics of the expression by passing the keyword argument `parser='python'`. This enforces the same semantics as evaluation in Python space. Likewise, you can pass `engine='python'` to evaluate an expression using Python itself as a backend. This is not recommended as it is inefficient compared to using `numexpr` as the engine.

The `DataFrame.index` and `DataFrame.columns` attributes of the `DataFrame` instance are placed in the query namespace by default, which allows you to treat both the index and columns of

the frame as a column in the frame. The identifier `index` is used for the frame index; you can also use the name of the index to identify it in a query. Please note that Python keywords may not be used as identifiers.

For further details and examples see the [query](#) documentation in *indexing*.

## Examples

```
>>> from numpy.random import randn
>>> from pandas import DataFrame
>>> df = pd.DataFrame(randn(10, 2), columns=list('ab'))
>>> df.query('a > b')
>>> df[df.a > df.b]  # same result as the previous expression
```

## pandas.DataFrame.radd

`DataFrame.radd(other, axis='columns', level=None, fill_value=None)`

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

### Returns

**result** [DataFrame]

**See also:**

[\*DataFrame.add\*](#)

## Notes

Mismatched indices will be unioned together

## Examples

```

>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0

```

## pandas.DataFrame.rank

`DataFrame.rank` (*axis=0*, *method='average'*, *numeric\_only=None*, *na\_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

**Parameters** *axis*: {0 or 'index', 1 or 'columns'}, default 0

index to direct ranking

**method**: {'average', 'min', 'max', 'first', 'dense'}

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**numeric\_only**: boolean, default None

Include only float, int, boolean data. Valid only for DataFrame or Panel objects

**na\_option**: {'keep', 'top', 'bottom'}

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

**ascending**: boolean, default True

False for ranks by high (1) to low (N)

**pct** : boolean, default False

Computes percentage rank of data

### Returns

**ranks** [same type as caller]

## pandas.DataFrame.rdiv

`DataFrame.rdiv(other, axis='columns', level=None, fill_value=None)`

Floating division of dataframe and other, element-wise (binary operator *rtruediv*).

Equivalent to `other / dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

### Returns

**result** [DataFrame]

### See also:

`DataFrame.truediv`

### Notes

Mismatched indices will be unioned together

### Examples

None

## pandas.DataFrame.reindex

`DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill_value=nan, limit=None, tolerance=None)`

Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`

**Parameters** **labels** : array-like, optional

New labels / index to conform the axis specified by 'axis' to.

**index, columns** : array-like, optional (should be specified using keywords)

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : int or str, optional

Axis to target. Can be either the axis name ('index', 'columns') or number (0, 1).

**method** : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

**copy** : boolean, default True

Return a new object, even if the passed indexes are the same

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value

**limit** : int, default None

Maximum number of consecutive elements to forward or backward fill

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation  $\text{abs}(\text{index}[\text{indexer}] - \text{target}) \leq \text{tolerance}$ .

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

**Returns**

**reindexed** [DataFrame]

## Examples

DataFrame.reindex supports two calling conventions

- (index=index\_labels, columns=column\_labels, ...)



- (labels, axis={'index', 'columns'}, ...)

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...     'http_status': [200, 200, 404, 404, 301],
...     'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...     index=index)
>>> df
```

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
```

	http_status	response_time
Safari	404.0	0.07
Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword method to fill the NaN values.

```
>>> df.reindex(new_index, fill_value=0)
```

	http_status	response_time
Safari	404	0.07
Iceweasel	0	0.00
Comodo Dragon	0	0.00
IE10	404	0.08
Chrome	200	0.02

```
>>> df.reindex(new_index, fill_value='missing')
```

	http_status	response_time
Safari	404	0.07
Iceweasel	missing	missing
Comodo Dragon	missing	missing
IE10	404	0.08
Chrome	200	0.02

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
```

	http_status	user_agent
Firefox	200	NaN
Chrome	200	NaN

(continues on next page)

(continued from previous page)

Safari	404	NaN
IE10	404	NaN
Konqueror	301	NaN

Or we can use “axis-style” keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
      http_status  user_agent
Firefox         200        NaN
Chrome          200        NaN
Safari          404        NaN
IE10            404        NaN
Konqueror       301        NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                     index=date_index)
>>> df2
      prices
2010-01-01    100
2010-01-02    101
2010-01-03     NaN
2010-01-04    100
2010-01-05     89
2010-01-06     88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
      prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01    100
2010-01-02    101
2010-01-03     NaN
2010-01-04    100
2010-01-05     89
2010-01-06     88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, ‘2009-12-29’) are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the method keyword.

```
>>> df2.reindex(date_index2, method='bfill')
      prices
2009-12-29    100
2009-12-30    100
2009-12-31    100
2010-01-01    100
```

(continues on next page)

(continued from previous page)

2010-01-02	101
2010-01-03	NaN
2010-01-04	100
2010-01-05	89
2010-01-06	88
2010-01-07	NaN

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the `fillna()` method.

See the [user guide](#) for more.

### pandas.DataFrame.reindex\_axis

`DataFrame.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=nan)`

Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`

**Parameters** `labels` : array-like

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : [{0 or 'index', 1 or 'columns'}]

**method** : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

**copy** : boolean, default True

Return a new object, even if the passed indexes are the same

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**limit** : int, default None

Maximum number of consecutive elements to forward or backward fill

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list,

tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

#### Returns

**reindexed** [DataFrame]

#### See also:

[\*reindex\*](#), [\*reindex\\_like\*](#)

#### Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

### pandas.DataFrame.reindex\_like

DataFrame.**reindex\_like** (*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices to myself.

#### Parameters

**other** [Object]

**method** [string or None]

**copy** [boolean, default True]

**limit** : int, default None

Maximum number of consecutive labels to fill for inexact matches.

**tolerance** : optional

Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

#### Returns

**reindexed** [same as input]

#### Notes

Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`

### pandas.DataFrame.rename

DataFrame.**rename** (*mapper=None*, *index=None*, *columns=None*, *axis=None*, *copy=True*, *inplace=False*, *level=None*)

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

See the [\*user guide\*](#) for more.

**Parameters** `mapper`, `index`, `columns` : dict-like or function, optional

dict-like or functions transformations to apply to that axis' values. Use either `mapper` and `axis` to specify the axis to target with `mapper`, or `index` and `columns`.

**axis** : int or str, optional

Axis to target with `mapper`. Can be either the axis name ('index', 'columns') or number (0, 1). The default is 'index'.

**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False

Whether to return a new DataFrame. If True then value of `copy` is ignored.

**level** : int or level name, default None

In case of a MultiIndex, only rename labels in the specified level.

**Returns**

**renamed** [DataFrame]

**See also:**

`pandas.DataFrame.rename_axis`

## Examples

`DataFrame.rename` supports two calling conventions

- `(index=index_mapper, columns=columns_mapper, ...)`
- `(mapper, axis={'index', 'columns'}, ...)`

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

## **pandas.DataFrame.rename\_axis**

`DataFrame.rename_axis` (*mapper*, *axis=0*, *copy=True*, *inplace=False*)

Alter the name of the index or columns.

**Parameters** **mapper** : scalar, list-like, optional

Value to set as the axis name attribute.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis.

**copy** : boolean, default True

Also copy underlying data.

**inplace** : boolean, default False

Modifies the object directly, instead of creating a new Series or DataFrame.

**Returns** **renamed** : Series, DataFrame, or None

The same type as the caller or None if *inplace* is True.

**See also:**

[`pandas.Series.rename`](#) Alter Series index labels or name

[`pandas.DataFrame.rename`](#) Alter DataFrame index labels or name

[`pandas.Index.rename`](#) Set new names on index

## **Notes**

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

## **Examples**

### **Series**

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

### **DataFrame**

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
```

	A	B
foo		
0	1	4
1	2	5
2	3	6

```
>>> df.rename_axis("bar", axis="columns")
```

	A	B
bar		
0	1	4
1	2	5
2	3	6

### pandas.DataFrame.reorder\_levels

`DataFrame.reorder_levels` (*order*, *axis=0*)

Rearrange index levels using input order. May not drop or duplicate levels

**Parameters** *order* : list of int or list of str

List representing new level order. Reference level by number (position) or by key (label).

**axis** : int

Where to reorder levels.

**Returns**

type of caller (new object)

### pandas.DataFrame.replace

`DataFrame.replace` (*to\_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to\_replace* with *value*.

Values of the DataFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

**Parameters** *to\_replace* : str, regex, list, dict, Series, int, float, or None

How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to *to\_replace* will be replaced with *value*
  - str: string exactly matching *to\_replace* will be replaced with *value*
  - regex: regexs matching *to\_replace* will be replaced with *value*
- list of str, regex, or numeric:
  - First, if *to\_replace* and *value* are both lists, they **must** be the same length.
  - Second, if *regex=True* then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter

much for *value* since there are only a few possible substitution regexes you can use.

- str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{ 'a': 'b', 'y': 'z' }` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{ 'a': 1, 'b': 'z' }` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be *None* in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{ 'a': { 'b': np.nan} }`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be *None* to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also *None* then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to replace any values matching *to\_replace* with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill.

**regex** : bool or same types as *to\_replace*, default False

Whether to interpret *to\_replace* and/or *value* as regular expressions. If this is True then *to\_replace* must be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to\_replace* must be *None*.

**method** : { 'pad', 'ffill', 'bfill', *None* }

The method to use when for replacement, when *to\_replace* is a scalar, list or tuple and *value* is *None*.

Changed in version 0.23.0: Added to DataFrame.

**Returns DataFrame**



Object after replacement.

#### Raises `AssertionError`

- If *regex* is not a `bool` and *to\_replace* is not `None`.

#### `TypeError`

- If *to\_replace* is a `dict` and *value* is not a `list`, `dict`, `ndarray`, or `Series`
- If *to\_replace* is `None` and *regex* is not compilable into a regular expression or is a `list`, `dict`, `ndarray`, or `Series`.
- When replacing multiple `bool` or `datetime64` objects and the arguments to *to\_replace* does not match the type of the value being replaced

#### `ValueError`

- If a `list` or an `ndarray` is passed to *to\_replace* and *value* but they are not the same length.

#### See also:

[`DataFrame.fillna`](#) Fill NA values

[`DataFrame.where`](#) Replace values based on boolean condition

[`Series.str.replace`](#) Simple string replacement.

#### Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.
- When `dict` is used as the *to\_replace* value, it is like `key(s)` in the `dict` are the *to\_replace* part and `value(s)` in the `dict` are the *value* parameter.

#### Examples

##### Scalar ‘to\_replace’ and ‘value’

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                     'B': [5, 6, 7, 8, 9],
...                     'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
```

	A	B	C
0	5	5	a
1	1	6	b
2	2	7	c
3	3	8	d
4	4	9	e

**List-like ‘to\_replace’**

```
>>> df.replace([0, 1, 2, 3], 4)
```

	A	B	C
0	4	5	a
1	4	6	b
2	4	7	c
3	4	8	d
4	4	9	e

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
```

	A	B	C
0	4	5	a
1	3	6	b
2	2	7	c
3	1	8	d
4	4	9	e

```
>>> s.replace([1, 2], method='bfill')
```

0	0
1	3
2	3
3	3
4	4

dtype: int64

**dict-like ‘to\_replace’**

```
>>> df.replace({0: 10, 1: 100})
```

	A	B	C
0	10	5	a
1	100	6	b
2	2	7	c
3	3	8	d
4	4	9	e

```
>>> df.replace({'A': 0, 'B': 5}, 100)
```

	A	B	C
0	100	100	a
1	1	6	b
2	2	7	c
3	3	8	d
4	4	9	e

```
>>> df.replace({'A': {0: 100, 4: 400}})
   A  B  C
0 100  5  a
1   1  6  b
2   2  7  c
3   3  8  d
4 400  9  e
```

**Regular expression ‘to\_replace’**

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                     'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
   A  B
0  new abc
1  foo new
2  bait xyz
```

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
   A  B
0  new abc
1  foo bar
2  bait xyz
```

```
>>> df.replace(regex=r'^ba.$', value='new')
   A  B
0  new abc
1  foo new
2  bait xyz
```

```
>>> df.replace(regex={'r'^ba.$': 'new', 'foo': 'xyz'})
   A  B
0  new abc
1  xyz new
2  bait xyz
```

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
   A  B
0  new abc
1  new new
2  bait xyz
```

Note that when replacing multiple bool or datetime64 objects, the data types in the *to\_replace* parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                     'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False}) # raises
Traceback (most recent call last):
...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the dict keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the *to\_replace* parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the *to\_replace* value, it is like the value(s) in the dict are equal to the *value* parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1     None
2     None
3        b
4     None
dtype: object
```

When *value=None* and *to\_replace* is a scalar, list or tuple, *replace* uses the *method* parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0      10
1      10
2      10
3        b
4        b
dtype: object
```

## pandas.DataFrame.resample

`DataFrame.resample` (*rule*, *how=None*, *axis=0*, *fill\_method=None*, *closed=None*, *label=None*, *convention='start'*, *kind=None*, *loffset=None*, *limit=None*, *base=0*, *on=None*, *level=None*)

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (DatetimeIndex, PeriodIndex, or TimedeltaIndex), or pass datetime-like values to the *on* or *level* keyword.

**Parameters** *rule* : string

the offset string or object representing target conversion

**axis** [int, optional, default 0]

**closed** : {'right', 'left'}

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**label** : {'right', 'left'}

Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**convention** : {'start', 'end', 's', 'e'}

For PeriodIndex only, controls whether to use the start or end of *rule*

**kind: {'timestamp', 'period'}, optional**

Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

**loffset : timedelta**

Adjust the resampled time labels

**base : int, default 0**

For frequencies that evenly subdivide 1 day, the “origin” of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

**on : string, optional**

For a `DataFrame`, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

**level : string or int, optional**

For a `MultiIndex`, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

**Returns****Resampler object****See also:**

[`groupby`](#) Group by mapping, function, label, or list of labels.

**Notes**

See the [user guide](#) for more.

To learn more about the offset strings, please see [this link](#).

**Examples**

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
```

(continues on next page)

(continued from previous page)

```
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket 2000-01-01 00:03:00 contains the value 3, but the summed value in the resampled bucket with the label 2000-01-01 00:03:00 does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00    3
2000-01-01 00:06:00   12
2000-01-01 00:09:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00    0
2000-01-01 00:03:00    6
2000-01-01 00:06:00   15
2000-01-01 00:09:00   15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00    1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the NaN values using the pad method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the NaN values using the bfill method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5
```

```
>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00     8
2000-01-01 00:03:00    17
2000-01-01 00:06:00    26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))

>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using ‘start’ *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using ‘end’ *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64
```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```
>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()
      a  b  c  d
time
2000-01-01 00:00:00  0  3  6  9
2000-01-01 00:03:00  0  3  6  9
2000-01-01 00:06:00  0  3  6  9
```

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```
>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                      columns=['a', 'b', 'c', 'd'],
                      index=pd.MultiIndex.from_product([time, [1, 2]]))
>>> df2.resample('3T', level=0).sum()
      a  b  c  d
time
2000-01-01 00:00:00  0  6 12 18
2000-01-01 00:03:00  0  4  8 12
```

## pandas.DataFrame.reset\_index

`DataFrame.reset_index(level=None, drop=False, inplace=False, col_level=0, col_fill="")`

For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level\_0', 'level\_1', etc. if any are None. For a standard index, the index name will be used (if set), otherwise a default 'index' or 'level\_0' (if 'index' is already taken) will be used.

**Parameters** `level` : int, str, tuple, or list, default None

Only remove the given levels from the index. Removes all levels by default

**drop** : boolean, default False

Do not try to insert index into dataframe columns. This resets the index to the default integer index.

**inplace** : boolean, default False

Modify the DataFrame in place (do not create a new object)

**col\_level** : int or str, default 0

If the columns have multiple levels, determines which level the labels are inserted into. By default it is inserted into the first level.

**col\_fill** : object, default ''

If the columns have multiple levels, determines how the other levels are named. If None then the index name is repeated.

**Returns**

**resetted** [DataFrame]



## Examples

```
>>> df = pd.DataFrame([('bird', 389.0),
...                     ('bird', 24.0),
...                     ('mammal', 80.5),
...                     ('mammal', np.nan)],
...                     index=['falcon', 'parrot', 'lion', 'monkey'],
...                     columns=('class', 'max_speed'))
>>> df
```

	class	max_speed
falcon	bird	389.0
parrot	bird	24.0
lion	mammal	80.5
monkey	mammal	NaN

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
   index  class  max_speed
0  falcon   bird    389.0
1  parrot   bird     24.0
2    lion  mammal     80.5
3  monkey  mammal     NaN
```

We can use the `drop` parameter to avoid the old index being added as a column:

```
>>> df.reset_index(drop=True)
   class  max_speed
0   bird    389.0
1   bird     24.0
2  mammal     80.5
3  mammal     NaN
```

You can also use `reset_index` with *MultiIndex*.

```
>>> index = pd.MultiIndex.from_tuples([('bird', 'falcon'),
...                                   ('bird', 'parrot'),
...                                   ('mammal', 'lion'),
...                                   ('mammal', 'monkey')],
...                                   names=['class', 'name'])
>>> columns = pd.MultiIndex.from_tuples([('speed', 'max'),
...                                      ('species', 'type')])
>>> df = pd.DataFrame([(389.0, 'fly'),
...                    (24.0, 'fly'),
...                    (80.5, 'run'),
...                    (np.nan, 'jump')],
...                    index=index,
...                    columns=columns)
>>> df
```

		speed	species
		max	type
class	name		
bird	falcon	389.0	fly
	parrot	24.0	fly
mammal	lion	80.5	run
	monkey	NaN	jump

If the index has multiple levels, we can reset a subset of them:

```
>>> df.reset_index(level='class')
      class  speed species
      max    type
name
falcon   bird  389.0    fly
parrot   bird   24.0    fly
lion     mammal  80.5    run
monkey   mammal   NaN    jump
```

If we are not dropping the index, by default, it is placed in the top level. We can place it in another level:

```
>>> df.reset_index(level='class', col_level=1)
      class  speed species
      max    type
name
falcon   bird  389.0    fly
parrot   bird   24.0    fly
lion     mammal  80.5    run
monkey   mammal   NaN    jump
```

When the index is inserted under another level, we can specify under which one with the parameter *col\_fill*:

```
>>> df.reset_index(level='class', col_level=1, col_fill='species')
      species  speed species
      class    max    type
name
falcon      bird  389.0    fly
parrot      bird   24.0    fly
lion        mammal  80.5    run
monkey      mammal   NaN    jump
```

If we specify a nonexistent level for *col\_fill*, it is created:

```
>>> df.reset_index(level='class', col_level=1, col_fill='genus')
      genus  speed species
      class    max    type
name
falcon      bird  389.0    fly
parrot      bird   24.0    fly
lion        mammal  80.5    run
monkey      mammal   NaN    jump
```

## pandas.DataFrame.rfloordiv

`DataFrame.rfloordiv(other, axis='columns', level=None, fill_value=None)`

Integer division of dataframe and other, element-wise (binary operator *rfloordiv*).

Equivalent to `other // dataframe`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

#### See also:

[`DataFrame.floordiv`](#)

#### Notes

Mismatched indices will be unioned together

#### Examples

None

### **pandas.DataFrame.rmod**

`DataFrame.rmod(other, axis='columns', level=None, fill_value=None)`

Modulo of dataframe and other, element-wise (binary operator *rmod*).

Equivalent to `other % dataframe`, but with support to substitute a `fill_value` for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

#### See also:

[`DataFrame.mod`](#)

## Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.rmul

`DataFrame.rmul` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Multiplication of dataframe and other, element-wise (binary operator *rmul*).

Equivalent to `other * dataframe`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

**See also:**

[`DataFrame.mul`](#)

## Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.rolling

`DataFrame.rolling` (*window*, *min\_periods*=None, *center*=False, *win\_type*=None, *on*=None, *axis*=0, *closed*=None)

Provides rolling window calculations.

New in version 0.18.0.

**Parameters** **window** : int, or offset

Size of the moving window. This is the number of observations used for calculating the statistic. Each window will be a fixed size.

If its an offset then this will be the time period of each window. Each window will be a variable sized based on the observations included in the time-period. This is only valid for datetimelike indexes. This is new in 0.19.0

**min\_periods** : int, default None

Minimum number of observations in window required to have a value (otherwise result is NA). For a window that is specified by an offset, this will default to 1.

**center** : boolean, default False

Set the labels at the center of the window.

**win\_type** : string, default None

Provide a window type. If None, all points are evenly weighted. See the notes below for further information.

**on** : string, optional

For a DataFrame, column on which to calculate the rolling window, rather than the index

**closed** : string, default None

Make the interval closed on the 'right', 'left', 'both' or 'neither' endpoints. For offset-based windows, it defaults to 'right'. For fixed windows, defaults to 'both'. Remaining cases not implemented for fixed windows.

New in version 0.20.0.

**axis** [int or string, default 0]

## Returns

**a Window or Rolling sub-classed for the particular operation**

See also:

***expanding*** Provides expanding transformations.

***ewm*** Provides exponential weighted functions

## Notes

By default, the result is set to the right edge of the window. This can be changed to the center of the window by setting `center=True`.

To learn more about the offsets & frequency strings, please see [this link](#).

The recognized win\_types are:

- boxcar
- triang
- blackman

- hamming
- bartlett
- parzen
- bohman
- blackmanharris
- nuttall
- barthann
- kaiser (needs beta)
- gaussian (needs std)
- general\_gaussian (needs power, width)
- slepian (needs width).

If `win_type=None` all points are evenly weighted. To learn more about different window types see [scipy.signal window functions](#).

## Examples

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]})
>>> df
   B
0  0.0
1  1.0
2  2.0
3  NaN
4  4.0
```

Rolling sum with a window length of 2, using the 'triang' window type.

```
>>> df.rolling(2, win_type='triang').sum()
   B
0  NaN
1  1.0
2  2.5
3  NaN
4  NaN
```

Rolling sum with a window length of 2, `min_periods` defaults to the window length.

```
>>> df.rolling(2).sum()
   B
0  NaN
1  1.0
2  3.0
3  NaN
4  NaN
```

Same as above, but explicitly set the `min_periods`

```
>>> df.rolling(2, min_periods=1).sum()
      B
0  0.0
1  1.0
2  3.0
3  2.0
4  4.0
```

A ragged (meaning not-a-regular frequency), time-indexed DataFrame

```
>>> df = pd.DataFrame({'B': [0, 1, 2, np.nan, 4]},
...                    index = [pd.Timestamp('20130101 09:00:00'),
...                              pd.Timestamp('20130101 09:00:02'),
...                              pd.Timestamp('20130101 09:00:03'),
...                              pd.Timestamp('20130101 09:00:05'),
...                              pd.Timestamp('20130101 09:00:06')])
```

```
>>> df
                B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  2.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

Contrasting to an integer rolling window, this will roll a variable length window corresponding to the time period. The default for `min_periods` is 1.

```
>>> df.rolling('2s').sum()
                B
2013-01-01 09:00:00  0.0
2013-01-01 09:00:02  1.0
2013-01-01 09:00:03  3.0
2013-01-01 09:00:05  NaN
2013-01-01 09:00:06  4.0
```

## pandas.DataFrame.round

`DataFrame.round(decimals=0, *args, **kwargs)`

Round a DataFrame to a variable number of decimal places.

**Parameters** `decimals` : int, dict, Series

Number of decimal places to round each column to. If an int is given, round each column to the same number of places. Otherwise dict and Series round to variable numbers of places. Column names should be in the keys if `decimals` is a dict-like, or in the index if `decimals` is a Series. Any columns not included in `decimals` will be left as is. Elements of `decimals` which are not columns of the input will be ignored.

**Returns**

**DataFrame object**

See also:

`numpy.around`, `Series.round`

## Examples

```
>>> df = pd.DataFrame(np.random.random([3, 3]),
...                    columns=['A', 'B', 'C'], index=['first', 'second', 'third'])
>>> df
           A          B          C
first  0.028208  0.992815  0.173891
second 0.038683  0.645646  0.577595
third   0.877076  0.149370  0.491027
>>> df.round(2)
           A          B          C
first   0.03   0.99   0.17
second  0.04   0.65   0.58
third   0.88   0.15   0.49
>>> df.round({'A': 1, 'C': 2})
           A          B          C
first   0.0  0.992815   0.17
second  0.0  0.645646   0.58
third   0.9  0.149370   0.49
>>> decimals = pd.Series([1, 0, 2], index=['A', 'B', 'C'])
>>> df.round(decimals)
           A  B          C
first   0.0  1   0.17
second  0.0  1   0.58
third   0.9  0   0.49
```

## pandas.DataFrame.rpow

`DataFrame.rpow` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Exponential power of dataframe and other, element-wise (binary operator *rpow*).

Equivalent to `other ** dataframe`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

### Returns

**result** [DataFrame]

See also:

[\*DataFrame.pow\*](#)



## Notes

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.rsub

`DataFrame.rsub` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

#### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

**See also:**

[`DataFrame.sub`](#)

## Notes

Mismatched indices will be unioned together

## Examples

```

>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN

```

(continues on next page)

(continued from previous page)

```

>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                         two=[3, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0 -3.0
b  1.0 -2.0
c  1.0  NaN
d -1.0  NaN
e  NaN -2.0

```

**pandas.DataFrame.rtruediv**`DataFrame.rtruediv` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)Floating division of dataframe and other, element-wise (binary operator *rtruediv*).Equivalent to `other / dataframe`, but with support to substitute a *fill\_value* for missing data in one of the inputs.**Parameters****other** [Series, DataFrame, or constant]**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

**Returns****result** [DataFrame]**See also:**`DataFrame.truediv`**Notes**

Mismatched indices will be unioned together

## Examples

None

### pandas.DataFrame.sample

`DataFrame.sample` (*n=None*, *frac=None*, *replace=False*, *weights=None*, *random\_state=None*, *axis=None*)

Return a random sample of items from an axis of object.

You can use *random\_state* for reproducibility.

**Parameters** *n* : int, optional

Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

**frac** : float, optional

Fraction of axis items to return. Cannot be used with *n*.

**replace** : boolean, optional

Sample with or without replacement. Default = False.

**weights** : str or ndarray-like, optional

Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when *axis* = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

**random\_state** : int or numpy.random.RandomState, optional

Seed for the random number generator (if int), or numpy RandomState object.

**axis** : int or string, optional

Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

### Returns

**A new object of same type as caller.**

## Examples

Generate an example Series and DataFrame:

```

>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0    -0.038497
1     1.820773
2    -0.972766
3    -1.598270

```

(continues on next page)

(continued from previous page)

```

4    -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()

```

	A	B	C	D
0	0.016443	-2.318952	-0.566372	-1.028078
1	-1.051921	0.438836	0.658280	-0.175797
2	-1.243569	-0.364626	-0.215065	0.057736
3	1.768216	0.404512	-0.385604	-1.457834
4	1.072446	-1.137172	0.314194	-0.046661

Next extract a random sample from both of these objects...

3 random elements from the Series:

```

>>> s.sample(n=3)
27    -0.994689
55    -1.049016
67    -0.224565
dtype: float64

```

And a random 10% of the DataFrame with replacement:

```

>>> df.sample(frac=0.1, replace=True)

```

	A	B	C	D
35	1.981780	0.142106	1.817165	-0.290805
49	-1.336199	-0.448634	-0.789640	0.217116
40	0.823173	-0.078816	1.009536	1.015108
15	1.421154	-0.055301	-1.922594	-0.019696
6	-0.148339	0.832938	1.787600	-1.383767

You can use *random state* for reproducibility:

```

>>> df.sample(random_state=1)

```

	A	B	C	D
37	-2.027662	0.103611	0.237496	-0.165867
43	-0.259323	-0.583426	1.516140	-0.479118
12	-1.686325	-0.579510	0.985195	-0.460286
8	1.167946	0.429082	1.215742	-1.636041
9	1.197475	-0.864188	1.554031	-1.505264

## pandas.DataFrame.select

`DataFrame.select (crit, axis=0)`

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use `df.loc[df.index.map(crit)]` to select via labels

**Parameters** `crit`: function

To be called on each index (label). Should return True or False

**axis** [int]

**Returns**

**selection** [type of caller]

**pandas.DataFrame.select\_dtypes**`DataFrame.select_dtypes` (*include=None, exclude=None*)

Return a subset of the DataFrame's columns based on the column dtypes.

**Parameters** `include, exclude` : scalar or list-like

A selection of dtypes or strings to be included/excluded. At least one of these parameters must be supplied.

**Returns** `subset` : DataFrameThe subset of the frame including the dtypes in `include` and excluding the dtypes in `exclude`.**Raises** `ValueError`

- If both of `include` and `exclude` are empty
- If `include` and `exclude` have overlapping elements
- If any kind of string dtype is passed in.

**Notes**

- To select all *numeric* types, use `np.number` or `'number'`
- To select strings you must use the `object` dtype, but note that this will return *all* object dtype columns
- See the [numpy dtype hierarchy](#)
- To select datetimes, use `np.datetime64`, `'datetime'` or `'datetime64'`
- To select timedeltas, use `np.timedelta64`, `'timedelta'` or `'timedelta64'`
- To select Pandas categorical dtypes, use `'category'`
- To select Pandas datetimetz dtypes, use `'datetimez'` (new in 0.20.0) or `'datetime64[ns, tz]'`

**Examples**

```
>>> df = pd.DataFrame({'a': [1, 2] * 3,
...                     'b': [True, False] * 3,
...                     'c': [1.0, 2.0] * 3})
>>> df
```

	a	b	c
0	1	True	1.0
1	2	False	2.0
2	1	True	1.0
3	2	False	2.0
4	1	True	1.0
5	2	False	2.0

```
>>> df.select_dtypes(include='bool')
b
0  True
```

(continues on next page)

(continued from previous page)

```
1 False
2 True
3 False
4 True
5 False
```

```
>>> df.select_dtypes(include=['float64'])
   c
0  1.0
1  2.0
2  1.0
3  2.0
4  1.0
5  2.0
```

```
>>> df.select_dtypes(exclude=['int'])
   b    c
0  True  1.0
1 False  2.0
2  True  1.0
3 False  2.0
4  True  1.0
5 False  2.0
```

## pandas.DataFrame.sem

DataFrame.**sem**(axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the ddof argument

### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**sem** [Series or DataFrame (if level specified)]

**pandas.DataFrame.set\_axis**

`DataFrame.set_axis` (*labels*, *axis*=0, *inplace*=None)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

**Parameters** *labels* : list-like, Index

The values for the new index.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

**inplace** : boolean, default None

Whether to return a new %(klass)s instance.

**Warning:** `inplace=None` currently falls back to `True`, but in a future version, will default to `False`. Use `inplace=True` explicitly rather than relying on the default.

**Returns** *renamed* : %(klass)s or None

An object of same type as caller if `inplace=False`, None otherwise.

**See also:**

[`pandas.DataFrame.rename\_axis`](#) Alter the name of the index or columns.

**Examples****Series**

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
2      3
dtype: int64
```

**DataFrame**

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
   A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
   I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
   i  ii
0  1   4
1  2   5
2  3   6
```

**pandas.DataFrame.set\_index**

`DataFrame.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)`

Set the DataFrame index (row labels) using one or more existing columns. By default yields a new object.

**Parameters**

**keys** [column label or list of column labels / arrays]

**drop** : boolean, default True

Delete columns to be used as the new index

**append** : boolean, default False

Whether to append columns to existing index

**inplace** : boolean, default False

Modify the DataFrame in place (do not create a new object)

**verify\_integrity** : boolean, default False

Check the new index for duplicates. Otherwise defer the check until necessary.

Setting to False will improve the performance of this method

**Returns**



**dataframe** [DataFrame]

## Examples

```
>>> df = pd.DataFrame({'month': [1, 4, 7, 10],
...                     'year': [2012, 2014, 2013, 2014],
...                     'sale': [55, 40, 84, 31]})
```

	month	sale	year
0	1	55	2012
1	4	40	2014
2	7	84	2013
3	10	31	2014

Set the index to become the ‘month’ column:

```
>>> df.set_index('month')
      sale  year
month
1      55  2012
4      40  2014
7      84  2013
10     31  2014
```

Create a multi-index using columns ‘year’ and ‘month’:

```
>>> df.set_index(['year', 'month'])
      sale
year month
2012  1    55
2014  4    40
2013  7    84
2014  10   31
```

Create a multi-index using a set of values and a column:

```
>>> df.set_index([1, 2, 3, 4], 'year')
      month  sale
year
1  2012    1    55
2  2014    4    40
3  2013    7    84
4  2014   10    31
```

## pandas.DataFrame.set\_value

`DataFrame.set_value(index, col, value, takeable=False)`

Put single value at passed column and index

Deprecated since version 0.21.0: Use `.at[]` or `.iat[]` accessors instead.

### Parameters

**index** [row label]**col** [column label]**value** [scalar value]

**takeable** [interpret the index/col as indexers, default False]

**Returns frame** : DataFrame

If label pair is contained, will be reference to calling DataFrame, otherwise a new object

## **pandas.DataFrame.shift**

DataFrame.**shift** (*periods=1, freq=None, axis=0*)

Shift index by desired number of periods with an optional time freq

**Parameters periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, optional

Increment to use from the tseries module or time rule (e.g. 'EOM'). See Notes.

**axis** [{0 or 'index', 1 or 'columns'}]

**Returns**

**shifted** [DataFrame]

## **Notes**

If freq is specified then the index values are shifted but the data is not realigned. That is, use freq if you would like to extend the index when shifting and preserve the original data.

## **pandas.DataFrame.skew**

DataFrame.**skew** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return unbiased skew over requested axis Normalized by N-1

**Parameters**

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**skew** [Series or DataFrame (if level specified)]

**pandas.DataFrame.slice\_shift**`DataFrame.slice_shift` (*periods=1, axis=0*)

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

**Parameters** *periods* : int

Number of periods to move, can be positive or negative

**Returns**

**shifted** [same type as caller]

**Notes**

While the *slice\_shift* is faster than *shift*, you may pay for it later during alignment.

**pandas.DataFrame.sort\_index**

`DataFrame.sort_index` (*axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na\_position='last', sort\_remaining=True, by=None*)

Sort object by labels (along an axis)

**Parameters**

**axis** [index, columns to direct sorting]

**level** : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

**ascending** : boolean, default True

Sort ascending vs. descending

**inplace** : bool, default False

if True, perform operation in-place

**kind** : { 'quicksort', 'mergesort', 'heapsort' }, default 'quicksort'

Choice of sorting algorithm. See also `ndarray.sort` for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

**na\_position** : { 'first', 'last' }, default 'last'

*first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

**sort\_remaining** : bool, default True

if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

**Returns**

**sorted\_obj** [DataFrame]

## pandas.DataFrame.sort\_values

`DataFrame.sort_values` (*by*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*,  
*na\_position='last'*)

Sort by the values along either axis

**Parameters** *by* : str or list of str

Name or list of names to sort by.

- if *axis* is 0 or *'index'* then *by* may contain index levels and/or column labels
- if *axis* is 1 or *'columns'* then *by* may contain column levels and/or index labels

Changed in version 0.23.0: Allow specifying index or column level names.

**axis** : {0 or *'index'*, 1 or *'columns'*}, default 0

Axis to be sorted

**ascending** : bool or list of bool, default True

Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the *by*.

**inplace** : bool, default False

if True, perform operation in-place

**kind** : {*'quicksort'*, *'mergesort'*, *'heapsort'*}, default *'quicksort'*

Choice of sorting algorithm. See also `ndarray.sort` for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

**na\_position** : {*'first'*, *'last'*}, default *'last'*

*first* puts NaNs at the beginning, *last* puts NaNs at the end

### Returns

**sorted\_obj** [DataFrame]

## Examples

```
>>> df = pd.DataFrame({
...     'col1' : ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2' : [2, 1, 9, 8, 7, 4],
...     'col3' : [0, 1, 9, 4, 2, 3],
... })
>>> df
   col1  col2  col3
0    A     2     0
1    A     1     1
2    B     9     9
3  NaN     8     4
4    D     7     2
5    C     4     3
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
   col1 col2 col3
0    A     2     0
1    A     1     1
2    B     9     9
5    C     4     3
4    D     7     2
3   NaN     8     4
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
   col1 col2 col3
1    A     1     1
0    A     2     0
2    B     9     9
5    C     4     3
4    D     7     2
3   NaN     8     4
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
   col1 col2 col3
4    D     7     2
5    C     4     3
2    B     9     9
0    A     2     0
1    A     1     1
3   NaN     8     4
```

Putting NAs first

```
>>> df.sort_values(by='col1', ascending=False, na_position='first')
   col1 col2 col3
3   NaN     8     4
4    D     7     2
5    C     4     3
2    B     9     9
0    A     2     0
1    A     1     1
```

## pandas.DataFrame.sortlevel

`DataFrame.sortlevel` (*level=0, axis=0, ascending=True, inplace=False, sort\_remaining=True*)

Sort multilevel index by chosen axis and primary level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order).

Deprecated since version 0.20.0: Use `DataFrame.sort_index()`

### Parameters

**level** [int]

**axis** [{0 or 'index', 1 or 'columns'}, default 0]

**ascending** [boolean, default True]

**inplace** : boolean, default False

Sort the DataFrame without creating a new instance

**sort\_remaining** : boolean, default True

Sort by the other levels too.

#### Returns

**sorted** [DataFrame]

**See also:**

`DataFrame.sort_index`

### **pandas.DataFrame.squeeze**

`DataFrame.squeeze` (*axis=None*)

Squeeze length 1 dimensions.

**Parameters** **axis** : None, integer or string axis name, optional

The axis to squeeze if 1-sized.

New in version 0.20.0.

#### Returns

**scalar if 1-sized, else original object**

### **pandas.DataFrame.stack**

`DataFrame.stack` (*level=-1, dropna=True*)

Stack the prescribed level(s) from columns to index.

Return a reshaped DataFrame or Series having a multi-level index with one or more new inner-most levels compared to the current DataFrame. The new inner-most levels are created by pivoting the columns of the current dataframe:

- if the columns have a single level, the output is a Series;
- if the columns have multiple levels, the new index level(s) is (are) taken from the prescribed level(s) and the output is a DataFrame.

The new index levels are sorted.

**Parameters** **level** : int, str, list, default -1

Level(s) to stack from the column axis onto the index axis, defined as one index or label, or a list of indices or labels.

**dropna** : bool, default True

Whether to drop rows in the resulting Frame/Series with missing values. Stacking a column level onto the index axis can create combinations of index and column values that are missing from the original dataframe. See Examples section.

**Returns** **DataFrame or Series**

Stacked dataframe or series.

**See also:**

**DataFrame.unstack** Unstack prescribed level(s) from index axis onto column axis.

**DataFrame.pivot** Reshape dataframe from long format to wide format.

**DataFrame.pivot\_table** Create a spreadsheet-style pivot table as a DataFrame.

## Notes

The function is named by analogy with a collection of books being re-organised from being side by side on a horizontal position (the columns of the dataframe) to being stacked vertically on top of each other (in the index of the dataframe).

## Examples

### Single level columns

```
>>> df_single_level_cols = pd.DataFrame([[0, 1], [2, 3]],
...                                     index=['cat', 'dog'],
...                                     columns=['weight', 'height'])
```

Stacking a dataframe with a single level column axis returns a Series:

```
>>> df_single_level_cols
   weight height
cat      0      1
dog      2      3
>>> df_single_level_cols.stack()
cat weight      0
   height      1
dog weight      2
   height      3
dtype: int64
```

### Multi level columns: simple case

```
>>> multicoll = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                       ('weight', 'pounds')])
>>> df_multi_level_cols1 = pd.DataFrame([[1, 2], [2, 4]],
...                                     index=['cat', 'dog'],
...                                     columns=multicoll)
```

Stacking a dataframe with a multi-level column axis:

```
>>> df_multi_level_cols1
   weight
   kg    pounds
cat   1         2
dog   2         4
>>> df_multi_level_cols1.stack()
   weight
cat kg      1
   pounds   2
dog kg      2
   pounds   4
```

### Missing values

```
>>> multicol2 = pd.MultiIndex.from_tuples([('weight', 'kg'),
...                                     ('height', 'm')])
>>> df_multi_level_cols2 = pd.DataFrame([[1.0, 2.0], [3.0, 4.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

It is common to have missing values when stacking a dataframe with multi-level columns, as the stacked dataframe typically has more values than the original dataframe. Missing values are filled with NaNs:

```
>>> df_multi_level_cols2
      weight height
      kg      m
cat    1.0    2.0
dog    3.0    4.0
>>> df_multi_level_cols2.stack()
      height weight
cat kg      NaN   1.0
   m      2.0   NaN
dog kg      NaN   3.0
   m      4.0   NaN
```

### Prescribing the level(s) to be stacked

The first parameter controls which level or levels are stacked:

```
>>> df_multi_level_cols2.stack(0)
      kg      m
cat height NaN  2.0
   weight 1.0 NaN
dog height NaN  4.0
   weight 3.0 NaN
>>> df_multi_level_cols2.stack([0, 1])
cat  height m      2.0
   weight kg      1.0
dog  height m      4.0
   weight kg      3.0
dtype: float64
```

### Dropping missing values

```
>>> df_multi_level_cols3 = pd.DataFrame([[None, 1.0], [2.0, 3.0]],
...                                     index=['cat', 'dog'],
...                                     columns=multicol2)
```

Note that rows where all values are missing are dropped by default but this behaviour can be controlled via the `dropna` keyword parameter:

```
>>> df_multi_level_cols3
      weight height
      kg      m
cat    NaN    1.0
dog    2.0    3.0
>>> df_multi_level_cols3.stack(dropna=False)
      height weight
cat kg      NaN   NaN
   m      1.0   NaN
dog kg      NaN   2.0
```

(continues on next page)



(continued from previous page)

```

      m      3.0      NaN
>>> df_multi_level_cols3.stack(dropna=True)
      height  weight
cat m      1.0      NaN
dog kg      NaN      2.0
      m      3.0      NaN

```

**pandas.DataFrame.std**

`DataFrame.std` (*axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs*)

Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the `ddof` argument

**Parameters**

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**std** [Series or DataFrame (if level specified)]

**pandas.DataFrame.sub**

`DataFrame.sub` (*other, axis='columns', level=None, fill\_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a `fill_value` for missing data in one of the inputs.

**Parameters**

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

### Returns

**result** [DataFrame]

### See also:

[`DataFrame.rsub`](#)

### Notes

Mismatched indices will be unioned together

### Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0 -3.0
b  1.0 -2.0
c  1.0  NaN
d -1.0  NaN
e  NaN -2.0
```

## pandas.DataFrame.subtract

`DataFrame.subtract` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

### Parameters

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

### Returns

**result** [DataFrame]

### See also:

[`DataFrame.rsub`](#)

### Notes

Mismatched indices will be unioned together

### Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                   columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                   index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0 -3.0
b  1.0 -2.0
c  1.0  NaN
d -1.0  NaN
e  NaN -2.0
```

## pandas.DataFrame.sum

`DataFrame.sum(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the sum of the values for the requested axis

### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

### Returns

**sum** [Series or DataFrame (if level specified)]

## Examples

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([]).sum() # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

**pandas.DataFrame.swapaxes**

`DataFrame.swapaxes` (*axis1*, *axis2*, *copy=True*)  
Interchange axes and swap values axes appropriately

**Returns**

**y** [same as input]

**pandas.DataFrame.swaplevel**

`DataFrame.swaplevel` (*i=-2*, *j=-1*, *axis=0*)  
Swap levels *i* and *j* in a MultiIndex on a particular axis

**Parameters** *i*, *j* : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

**Returns**

**swapped** [type of caller (new object)]

**.. versionchanged:: 0.18.1**

The indexes *i* and *j* are now optional, and default to the two innermost levels of the index.

**pandas.DataFrame.tail**

`DataFrame.tail` (*n=5*)  
Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

**Parameters** *n* : int, default 5

Number of rows to select.

**Returns** type of caller

The last *n* rows of the caller object.

**See also:**

[`pandas.DataFrame.head`](#) The first *n* rows of the caller object.

**Examples**

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0 alligator
1      bee
2    falcon
3      lion
```

(continues on next page)

(continued from previous page)

```
4    monkey
5    parrot
6     shark
7     whale
8     zebra
```

Viewing the last 5 lines

```
>>> df.tail()
      animal
4    monkey
5    parrot
6     shark
7     whale
8     zebra
```

Viewing the last  $n$  lines (three in this case)

```
>>> df.tail(3)
      animal
6     shark
7     whale
8     zebra
```

## pandas.DataFrame.take

`DataFrame.take` (*indices*, *axis=0*, *convert=None*, *is\_copy=True*, *\*\*kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters** **indices** : array-like

An array of ints indicating which positions to take.

**axis** : {0 or 'index', 1 or 'columns', None}, default 0

The axis on which to select elements. 0 means that we are selecting rows, 1 means that we are selecting columns.

**convert** : bool, default True

Whether to convert negative indices into positive ones. For example, -1 would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

**is\_copy** : bool, default True

Whether to return a copy of the original object or not.

**\*\*kwargs**

For compatibility with `numpy.take()`. Has no effect on the output.

**Returns** **taken** : type of caller

An array-like containing the elements taken from the object.

**See also:**

**DataFrame.loc** Select a subset of a DataFrame by labels.

**DataFrame.iloc** Select a subset of a DataFrame by positions.

**numpy.take** Take elements from an array along an axis.

## Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                    columns=['name', 'class', 'max_speed'],
...                    index=[0, 2, 3, 1])
>>> df
```

	name	class	max_speed
0	falcon	bird	389.0
2	parrot	bird	24.0
3	lion	mammal	80.5
1	monkey	mammal	NaN

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
```

	name	class	max_speed
0	falcon	bird	389.0
1	monkey	mammal	NaN

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
```

	class	max_speed
0	bird	389.0
2	bird	24.0
3	mammal	80.5
1	mammal	NaN

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
```

	name	class	max_speed
1	monkey	mammal	NaN
3	lion	mammal	80.5

## pandas.DataFrame.to\_clipboard

**DataFrame.to\_clipboard** (*excel=True, sep=None, \*\*kwargs*)

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

**Parameters** `excel` : bool, default True

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

**sep** : str, default ' \t '

Field delimiter.

**\*\*kwargs**

These parameters will be passed to `DataFrame.to_csv`.

**See also:**

[`DataFrame.to\_csv`](#) Write a `DataFrame` to a comma-separated values (csv) file.

[`read\_clipboard`](#) Read text from clipboard and pass to `read_table`.

## Notes

Requirements for your platform.

- Linux : `xclip`, or `xsel` (with `gtk` or `PyQt4` modules)
- Windows : none
- OS X : none

## Examples

Copy the contents of a `DataFrame` to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword `index` and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```



**pandas.DataFrame.to\_csv**

```
DataFrame.to_csv(path_or_buf=None, sep=', ', na_rep="", float_format=None, columns=None,
                  header=True, index=True, index_label=None, mode='w', encoding=None,
                  compression=None, quoting=None, quotechar='"', line_terminator='\n',
                  chunksize=None, tupleize_cols=None, date_format=None, doublequote=True,
                  escapechar=None, decimal='.')
```

Write DataFrame to a comma-separated values (csv) file

**Parameters** **path\_or\_buf** : string or file handle, default None

File path or object, if None is provided the result is returned as a string.

**sep** : character, default ','

Field delimiter for the output file.

**na\_rep** : string, default ''

Missing data representation

**float\_format** : string, default None

Format string for floating point numbers

**columns** : sequence, optional

Columns to write

**header** : boolean or list of string, default True

Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, or False, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. If False do not print fields for index names. Use *index\_label=False* for easier importing in R

**mode** : str

Python write mode, default 'w'

**encoding** : string, optional

A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

**compression** : string, optional

A string representing the compression to use in the output file. Allowed values are 'gzip', 'bz2', 'zip', 'xz'. This input is only used when the first argument is a filename.

**line\_terminator** : string, default '\n'

The newline character or character sequence to use in the output file

**quoting** : optional constant from csv module

defaults to `csv.QUOTE_MINIMAL`. If you have set a *float\_format* then floats are converted to strings and thus `csv.QUOTE_NONNUMERIC` will treat them as non-numeric

**quotechar** : string (length 1), default `“”`

character used to quote fields

**doublequote** : boolean, default `True`

Control quoting of *quotechar* inside a field

**escapechar** : string (length 1), default `None`

character used to escape *sep* and *quotechar* when appropriate

**chunksize** : int or `None`

rows to write at a time

**tupleize\_cols** : boolean, default `False`

Deprecated since version 0.21.0: This argument will be removed and will always write each row of the multi-index as a separate row in the CSV file.

Write MultiIndex columns as a list of tuples (if `True`) or in the new, expanded format, where each MultiIndex column is a row in the CSV (if `False`).

**date\_format** : string, default `None`

Format string for datetime objects

**decimal**: string, default `‘.`

Character recognized as decimal separator. E.g. use `‘,’` for European data

## **pandas.DataFrame.to\_dense**

`DataFrame.to_dense()`

Return dense representation of NDFrame (as opposed to sparse)

## **pandas.DataFrame.to\_dict**

`DataFrame.to_dict(orient='dict', into=<class 'dict'>)`

Convert the DataFrame to a dictionary.

The type of the key-value pairs can be customized with the parameters (see below).

**Parameters** **orient** : str {`‘dict’`, `‘list’`, `‘series’`, `‘split’`, `‘records’`, `‘index’`}

Determines the type of the values of the dictionary.

- `‘dict’` (default) : dict like {column -> {index -> value}}
- `‘list’` : dict like {column -> [values]}
- `‘series’` : dict like {column -> Series(values)}
- `‘split’` : dict like {`‘index’` -> [index], `‘columns’` -> [columns], `‘data’` -> [values]}
- `‘records’` : list like [{column -> value}, ... , {column -> value}]
- `‘index’` : dict like {index -> {column -> value}}

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

**into** : class, default dict

The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

### Returns

**result** [collections.Mapping like {column -> {index -> value}}]

See also:

[`DataFrame.from\_dict`](#) create a DataFrame from a dictionary

[`DataFrame.to\_json`](#) convert a DataFrame to JSON format

### Examples

```
>>> df = pd.DataFrame({'col1': [1, 2],
...                     'col2': [0.5, 0.75]},
...                     index=['a', 'b'])
>>> df
   col1  col2
a      1   0.50
b      2   0.75
>>> df.to_dict()
{'col1': {'a': 1, 'b': 2}, 'col2': {'a': 0.5, 'b': 0.75}}
```

You can specify the return orientation.

```
>>> df.to_dict('series')
{'col1': a      1
         b      2
         Name: col1, dtype: int64,
 'col2': a      0.50
         b      0.75
         Name: col2, dtype: float64}
```

```
>>> df.to_dict('split')
{'index': ['a', 'b'], 'columns': ['col1', 'col2'],
 'data': [[1.0, 0.5], [2.0, 0.75]]}
```

```
>>> df.to_dict('records')
[{'col1': 1.0, 'col2': 0.5}, {'col1': 2.0, 'col2': 0.75}]
```

```
>>> df.to_dict('index')
{'a': {'col1': 1.0, 'col2': 0.5}, 'b': {'col1': 2.0, 'col2': 0.75}}
```

You can also specify the mapping type.

```
>>> from collections import OrderedDict, defaultdict
>>> df.to_dict(into=OrderedDict)
OrderedDict([('col1', OrderedDict([('a', 1), ('b', 2)])),
            ('col2', OrderedDict([('a', 0.5), ('b', 0.75)]))])
```

If you want a *defaultdict*, you need to initialize it:

```
>>> dd = defaultdict(list)
>>> df.to_dict('records', into=dd)
[defaultdict(<class 'list'>, {'col1': 1.0, 'col2': 0.5}),
 defaultdict(<class 'list'>, {'col1': 2.0, 'col2': 0.75})]
```

## pandas.DataFrame.to\_excel

`DataFrame.to_excel(excel_writer, sheet_name='Sheet1', na_rep="", float_format=None, columns=None, header=True, index=True, index_label=None, startrow=0, startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf', verbose=True, freeze_panes=None)`

Write DataFrame to an excel sheet

**Parameters excel\_writer** : string or ExcelWriter object

File path or existing ExcelWriter

**sheet\_name** : string, default 'Sheet1'

Name of sheet which will contain DataFrame

**na\_rep** : string, default ''

Missing data representation

**float\_format** : string, default None

Format string for floating point numbers

**columns** : sequence, optional

Columns to write

**header** : boolean or list of string, default True

Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**startrow** :

upper left cell row to dump data frame

**startcol** :

upper left cell column to dump data frame

**engine** : string, default None

write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**merge\_cells** : boolean, default True

Write MultiIndex and Hierarchical Rows as merged cells.

**encoding: string, default None**

encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

**inf\_rep : string, default 'inf'**

Representation for infinity (there is no native representation for infinity in Excel)

**freeze\_panes : tuple of integer (length 2), default None**

Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

## Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer, 'Sheet1')
>>> df2.to_excel(writer, 'Sheet2')
>>> writer.save()
```

For compatibility with to\_csv, to\_excel serializes lists and dicts to strings before writing.

## pandas.DataFrame.to\_feather

`DataFrame.to_feather(fname)`

write out the binary feather-format for DataFrames

New in version 0.20.0.

**Parameters fname : str**

string file path

## pandas.DataFrame.to\_gbq

`DataFrame.to_gbq(destination_table, project_id, chunksize=None, verbose=None, reauth=False, if_exists='fail', private_key=None, auth_local_webserver=False, table_schema=None)`

Write a DataFrame to a Google BigQuery table.

This function requires the `pandas-gbq` package.

Authentication to the Google BigQuery service is via OAuth 2.0.

- If `private_key` is provided, the library loads the JSON service account credentials and uses those to authenticate.
- If no `private_key` is provided, the library tries `application default credentials`.
- If application default credentials are not found or cannot be used with BigQuery, the library authenticates with user account credentials. In this case, you will be asked to grant permissions for product name 'pandas GBQ'.

**Parameters** `destination_table` : str

Name of table to be written, in the form 'dataset.tablename'.

**project\_id** : str

Google BigQuery Account project ID.

**chunksize** : int, optional

Number of rows to be inserted in each chunk from the dataframe. Set to `None` to load the whole dataframe at once.

**reauth** : bool, default False

Force Google BigQuery to reauthenticate the user. This is useful if multiple accounts are used.

**if\_exists** : str, default 'fail'

Behavior when the destination table exists. Value can be one of:

'fail' If table exists, do nothing.

'replace' If table exists, drop it, recreate it, and insert data.

'append' If table exists, insert data. Create if does not exist.

**private\_key** : str, optional

Service account private key in JSON format. Can be file path or string contents. This is useful for remote server authentication (eg. Jupyter/IPython notebook on remote host).

**auth\_local\_webserver** : bool, default False

Use the [local webserver flow](#) instead of the [console flow](#) when getting user credentials.

*New in version 0.2.0 of pandas-gbq.*

**table\_schema** : list of dicts, optional

List of BigQuery table fields to which according DataFrame columns conform to, e.g. `[{'name': 'col1', 'type': 'STRING'}, ...]`. If schema is not provided, it will be generated according to dtypes of DataFrame columns. See BigQuery API documentation on available names of a field.

*New in version 0.3.1 of pandas-gbq.*

**verbose** : boolean, deprecated

*Deprecated in Pandas-GBQ 0.4.0.* Use the [logging module](#) to adjust verbosity instead.

**See also:**

[pandas\\_gbq.to\\_gbq](#) This function in the pandas-gbq library.

[pandas.read\\_gbq](#) Read a DataFrame from Google BigQuery.

**pandas.DataFrame.to\_hdf**

`DataFrame.to_hdf` (*path\_or\_buf*, *key*, *\*\*kwargs*)

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the [user guide](#).

**Parameters** `path_or_buf` : str or pandas.HDFStore

File path or HDFStore object.

**key** : str

Identifier for the group in the store.

**mode** : {'a', 'w', 'r+'}, default 'a'

Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

**format** : {'fixed', 'table'}, default 'fixed'

Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

**append** : bool, default False

For Table formats, append the input data to the existing.

**data\_columns** : list of columns or True, optional

List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See [Query via Data Columns](#). Applicable only to format='table'.

**complevel** : {0-9}, optional

Specifies a compression level for data. A value of 0 disables compression.

**complib** : {'zlib', 'lzo', 'bzip2', 'blosc'}, default 'zlib'

Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): {'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd'}. Specifying a compression library which is not available issues a ValueError.

**fletcher32** : bool, default False

If applying compression use the fletcher32 checksum.

**dropna** : bool, default False

If true, ALL nan rows will not be written to store.

**errors** : str, default 'strict'

Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

See also:

**DataFrame.read\_hdf** Read from HDF file.

**DataFrame.to\_parquet** Write a DataFrame to the binary parquet format.

**DataFrame.to\_sql** Write to a sql table.

**DataFrame.to\_feather** Write out feather-format for DataFrames.

**DataFrame.to\_csv** Write out to a csv file.

## Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                    index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```



**pandas.DataFrame.to\_html**

`DataFrame.to_html` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, justify=None, bold\_rows=True, classes=None, escape=True, max\_rows=None, max\_cols=None, show\_dimensions=False, notebook=False, decimal='.', border=None, table\_id=None*)

Render a DataFrame as an HTML table.

*to\_html*-specific options:

**bold\_rows** [boolean, default True] Make the row labels bold in the output

**classes** [str or list or tuple, default None] CSS class(es) to apply to the resulting html table

**escape** [boolean, default True] Convert the characters <, >, and & to HTML-safe sequences.

**max\_rows** [int, optional] Maximum number of rows to show before truncating. If None, show all.

**max\_cols** [int, optional] Maximum number of columns to show before truncating. If None, show all.

**decimal** [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe

New in version 0.18.0.

**border** [int] A `border=border` attribute is included in the opening `<table>` tag. Default `pd.options.html.border`.

New in version 0.19.0.

**table\_id** [str, optional] A css id is included in the opening `<table>` tag if specified.

New in version 0.23.0.

**Parameters** **buf**: StringIO-like, optional

buffer to write to

**columns**: sequence, optional

the subset of columns to write; default None writes all columns

**col\_space**: int, optional

the minimum width of each column

**header**: bool, optional

whether to print column labels, default True

**index**: bool, optional

whether to print index (row) labels, default True

**na\_rep**: string, optional

string representation of NAN to use, default 'NaN'

**formatters**: list or dict of one-parameter functions, optional

formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

**float\_format**: one-parameter function, optional

formatter function to apply to columns' elements if they are floats, default None.  
The result of this function must be a unicode string.

**sparsify** : bool, optional

Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

**index\_names** : bool, optional

Prints the names of the indexes, default True

**line\_width** : int, optional

Width to wrap a line in characters, default no wrap

**table\_id** : str, optional

id for the <table> element create by to\_html

New in version 0.23.0.

**justify** : str, default None

How to justify the column labels. If None uses the option from the print configuration (controlled by set\_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

#### Returns

**formatted** [string (or unicode, depending on data and options)]

### pandas.DataFrame.to\_json

`DataFrame.to_json` (*path\_or\_buf=None, orient=None, date\_format=None, double\_precision=10, force\_ascii=True, date\_unit='ms', default\_handler=None, lines=False, compression=None, index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

**Parameters** **path\_or\_buf** : string or file handle, optional

File path or object. If not specified, the result is returned as a string.

**orient** : string

Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values'}
- The format of the JSON string
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, ... , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}} describing the data, and the data component is like `orient='records'`.

Changed in version 0.20.0.

**date\_format** : {None, 'epoch', 'iso'}

Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

**double\_precision** : int, default 10

The number of decimal places to use when encoding floating point values.

**force\_ascii** : boolean, default True

Force encoded string to be ASCII.

**date\_unit** : string, default 'ms' (milliseconds)

The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

**default\_handler** : callable, default None

Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

**lines** : boolean, default False

If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

**compression** : {None, 'gzip', 'bz2', 'zip', 'xz'}

A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

**index** : boolean, default True

Whether to include the index values in the JSON string. Not including the index (index=False) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

**See also:**

[`pandas.read\_json`](#)

## Examples

```
>>> df = pd.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                    index=[ 'row 1', 'row 2'],
...                    columns=[ 'col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns":["col 1","col 2"],
  "index":["row 1","row 2"],
  "data":[["a","b"],["c","d"]]]'
```

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1":"a","col 2":"b"}, {"col 1":"c","col 2":"d"}]'
```

Encoding/decoding a Dataframe using 'index' formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1":{"col 1":"a","col 2":"b"},"row 2":{"col 1":"c","col 2":"d"}}'
```

Encoding/decoding a Dataframe using 'columns' formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1":{"row 1":"a","row 2":"c"},"col 2":{"row 1":"b","row 2":"d"}}'
```

Encoding/decoding a Dataframe using 'values' formatted JSON:

```
>>> df.to_json(orient='values')
'[["a","b"],["c","d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
                        {"name": "col 2", "type": "string"}],
  "primaryKey": "index",
  "pandas_version": "0.20.0"},
  "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
            {"index": "row 2", "col 1": "c", "col 2": "d"}]}'
```

**pandas.DataFrame.to\_latex**

`DataFrame.to_latex` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, bold\_rows=False, column\_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multicolumn\_format=None, multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires `\usepackage{booktabs}`.

Changed in version 0.20.2: Added to Series

*to\_latex*-specific options:

**bold\_rows** [boolean, default False] Make the row labels bold in the output

**column\_format** [str, default None] The columns format as specified in [LaTeX table format](#) e.g 'rcl' for 3 columns

**longtable** [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a `\usepackage{longtable}` to your LaTeX preamble.

**escape** [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

**encoding** [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

**decimal** [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

**multicolumn** [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

**multicolumn\_format** [str, default 'l'] The alignment for multicolumns, similar to *column\_format* The default will be read from the config module.

New in version 0.20.0.

**multirow** [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a `\usepackage{multirow}` to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.

**pandas.DataFrame.to\_msgpack**

`DataFrame.to_msgpack` (*path\_or\_buf=None, encoding='utf-8', \*\*kwargs*)  
msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

**Parameters** **path** : string File path, buffer-like, or None

if None, return generated string

**append** : boolean whether to append to an existing msgpack

(default is False)

**compress** : type of compressor (zlib or blosc), default to None (no compression)

### **pandas.DataFrame.to\_panel**

`DataFrame.to_panel()`

Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Deprecated since version 0.20.0.

Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

#### **Returns**

**panel** [Panel]

### **pandas.DataFrame.to\_parquet**

`DataFrame.to_parquet(fname, engine='auto', compression='snappy', **kwargs)`

Write a DataFrame to the binary parquet format.

New in version 0.21.0.

This function writes the dataframe as a [parquet file](#). You can choose different parquet backends, and have the option of compression. See [the user guide](#) for more details.

#### **Parameters fname** : str

String file path.

**engine** : {'auto', 'pyarrow', 'fastparquet'}, default 'auto'

Parquet library to use. If 'auto', then the option `io.parquet.engine` is used. The default `io.parquet.engine` behavior is to try 'pyarrow', falling back to 'fastparquet' if 'pyarrow' is unavailable.

**compression** : {'snappy', 'gzip', 'brotli', None}, default 'snappy'

Name of the compression to use. Use `None` for no compression.

#### **\*\*kwargs**

Additional arguments passed to the parquet library. See [pandas io](#) for more details.

#### **See also:**

[read\\_parquet](#) Read a parquet file.

[DataFrame.to\\_csv](#) Write a csv file.

[DataFrame.to\\_sql](#) Write to a sql table.

[DataFrame.to\\_hdf](#) Write to hdf.

#### **Notes**

This function requires either the [fastparquet](#) or [pyarrow](#) library.

## Examples

```
>>> df = pd.DataFrame(data={'col1': [1, 2], 'col2': [3, 4]})
>>> df.to_parquet('df.parquet.gz', compression='gzip')
>>> pd.read_parquet('df.parquet.gz')
   col1  col2
0     1     3
1     2     4
```

## pandas.DataFrame.to\_period

`DataFrame.to_period` (*freq=None, axis=0, copy=True*)

Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)

### Parameters

**freq** [string, default]

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

**copy** : boolean, default True

If False then underlying input data is not copied

### Returns

**ts** [TimeSeries with PeriodIndex]

## pandas.DataFrame.to\_pickle

`DataFrame.to_pickle` (*path, compression='infer', protocol=4*)

Pickle (serialize) object to file.

### Parameters

**path** : str

File path where the pickled object will be stored.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

**protocol** : int

Int which indicates which protocol should be used by the pickler, default HIGHEST\_PROTOCOL (see [R15] paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python >= 3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST\_PROTOCOL.

New in version 0.21.0.

See also:

`read_pickle` Load pickled pandas object (or any object) from file.

`DataFrame.to_hdf` Write DataFrame to an HDF5 file.

`DataFrame.to_sql` Write DataFrame to a SQL database.

`DataFrame.to_parquet` Write a DataFrame to the binary parquet format.

## Examples

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

## pandas.DataFrame.to\_records

`DataFrame.to_records` (*index=True, convert\_datetime64=None*)

Convert DataFrame to a NumPy record array.

Index will be put in the ‘index’ field of the record array if requested.

**Parameters** `index` : boolean, default True

Include index in resulting record array, stored in ‘index’ field.

**convert\_datetime64** : boolean, default None

Deprecated since version 0.23.0.

Whether to convert the index to `datetime.datetime` if it is a `DatetimeIndex`.

**Returns**

`y` [numpy.recarray]

**See also:**

`DataFrame.from_records` convert structured or record ndarray to DataFrame.

`numpy.recarray` ndarray that allows field access using attributes, analogous to typed columns in a spreadsheet.



## Examples

```
>>> df = pd.DataFrame({'A': [1, 2], 'B': [0.5, 0.75]},
...                    index=['a', 'b'])
>>> df
   A    B
a  1  0.5
b  2  0.75
>>> df.to_records()
rec.array([( 'a', 1, 0.5 ), ( 'b', 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The index can be excluded from the record array:

```
>>> df.to_records(index=False)
rec.array([(1, 0.5 ), (2, 0.75)],
          dtype=[('A', '<i8'), ('B', '<f8')])
```

By default, timestamps are converted to *datetime.datetime*:

```
>>> df.index = pd.date_range('2018-01-01 09:00', periods=2, freq='min')
>>> df
              A    B
2018-01-01 09:00:00  1  0.50
2018-01-01 09:01:00  2  0.75
>>> df.to_records()
rec.array([(datetime.datetime(2018, 1, 1, 9, 0), 1, 0.5 ),
          (datetime.datetime(2018, 1, 1, 9, 1), 2, 0.75)],
          dtype=[('index', 'O'), ('A', '<i8'), ('B', '<f8')])
```

The timestamp conversion can be disabled so NumPy's *datetime64* data type is used instead:

```
>>> df.to_records(convert_datetime64=False)
rec.array([( '2018-01-01T09:00:00.000000000', 1, 0.5 ),
          ( '2018-01-01T09:01:00.000000000', 2, 0.75)],
          dtype=[('index', '<M8[ns]'), ('A', '<i8'), ('B', '<f8')])
```

## pandas.DataFrame.to\_sparse

`DataFrame.to_sparse` (*fill\_value=None, kind='block'*)

Convert to SparseDataFrame

### Parameters

**fill\_value** [float, default NaN]

**kind** [{ 'block', 'integer' }]

### Returns

**y** [SparseDataFrame]

**pandas.DataFrame.to\_stata**

`DataFrame.to_stata` (*fname*, *convert\_dates=None*, *write\_index=True*, *encoding='latin-1'*, *byteorder=None*, *time\_stamp=None*, *data\_label=None*, *variable\_labels=None*, *version=114*, *convert\_strl=None*)

Export Stata binary dta files.

**Parameters** **fname** : path (string), buffer or path object

string, path object (`pathlib.Path` or `py._path.local.LocalPath`) or object implementing a binary `write()` functions. If using a buffer then the buffer will not be automatically closed after the file data has been written.

**convert\_dates** : dict

Dictionary mapping columns containing datetime types to stata internal format to use when writing the dates. Options are 'tc', 'td', 'tm', 'tw', 'th', 'tq', 'ty'. Column can be either an integer or a name. Datetime columns that do not have a conversion type specified will be converted to 'tc'. Raises `NotImplementedError` if a datetime column has timezone information.

**write\_index** : bool

Write the index to Stata dataset.

**encoding** : str

Default is latin-1. Unicode is not supported.

**byteorder** : str

Can be ">", "<", "little", or "big". default is `sys.byteorder`.

**time\_stamp** : datetime

A datetime to use as file creation date. Default is the current time.

**data\_label** : str

A label for the data set. Must be 80 characters or smaller.

**variable\_labels** : dict

Dictionary containing columns as keys and variable labels as values. Each label must be 80 characters or smaller.

New in version 0.19.0.

**version** : {114, 117}

Version to use in the output dta file. Version 114 can be used read by Stata 10 and later. Version 117 can be read by Stata 13 or later. Version 114 limits string variables to 244 characters or fewer while 117 allows strings with lengths up to 2,000,000 characters.

New in version 0.23.0.

**convert\_strl** : list, optional

List of column names to convert to string columns to Stata StrL format. Only available if version is 117. Storing strings in the StrL format can produce smaller dta files if strings have more than 8 characters and values are repeated.

New in version 0.23.0.

**Raises** `NotImplementedError`

- If datetimes contain timezone information
- Column dtype is not representable in Stata

**ValueError**

- Columns listed in `convert_dates` are neither `datetime64[ns]` or `datetime.datetime`
- Column listed in `convert_dates` is not in `DataFrame`
- Categorical label contains more than 32,000 characters

New in version 0.19.0.

**See also:**

`pandas.read_stata` Import Stata data files

`pandas.io.stata.StataWriter` low-level writer for Stata data files

`pandas.io.stata.StataWriter117` low-level writer for version 117 files

**Examples**

```
>>> data.to_stata('./data_file.dta')
```

Or with dates

```
>>> data.to_stata('./date_data_file.dta', {2 : 'tw'})
```

Alternatively you can create an instance of the `StataWriter` class

```
>>> writer = StataWriter('./data_file.dta', data)
>>> writer.write_file()
```

With dates:

```
>>> writer = StataWriter('./date_data_file.dta', data, {2 : 'tw'})
>>> writer.write_file()
```

**pandas.DataFrame.to\_string**

`DataFrame.to_string`(*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, justify=None, line\_width=None, max\_rows=None, max\_cols=None, show\_dimensions=False*)

Render a `DataFrame` to a console-friendly tabular output.

**Parameters** `buf` : StringIO-like, optional

buffer to write to

**columns** : sequence, optional

the subset of columns to write; default `None` writes all columns

**col\_space** : int, optional

the minimum width of each column

**header** : bool, optional

Write out the column names. If a list of strings is given, it is assumed to be aliases for the column names

**index** : bool, optional

whether to print index (row) labels, default True

**na\_rep** : string, optional

string representation of NAN to use, default 'NaN'

**formatters** : list or dict of one-parameter functions, optional

formatter functions to apply to columns' elements by position or name, default None. The result of each function must be a unicode string. List must be of length equal to the number of columns.

**float\_format** : one-parameter function, optional

formatter function to apply to columns' elements if they are floats, default None. The result of this function must be a unicode string.

**sparsify** : bool, optional

Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

**index\_names** : bool, optional

Prints the names of the indexes, default True

**line\_width** : int, optional

Width to wrap a line in characters, default no wrap

**table\_id** : str, optional

id for the <table> element create by to\_html

New in version 0.23.0.

**justify** : str, default None

How to justify the column labels. If None uses the option from the print configuration (controlled by set\_option), 'right' out of the box. Valid values are

- left
- right
- center
- justify
- justify-all
- start
- end
- inherit
- match-parent
- initial
- unset

**Returns****formatted** [string (or unicode, depending on data and options)]**pandas.DataFrame.to\_timestamp**`DataFrame.to_timestamp` (*freq=None, how='start', axis=0, copy=True*)Cast to DatetimeIndex of timestamps, at *beginning* of period**Parameters** **freq** : string, default frequency of PeriodIndex

Desired frequency

**how** : {'s', 'e', 'start', 'end'}

Convention for converting period to timestamp; start of period vs. end

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to convert (the index by default)

**copy** : boolean, default True

If false then underlying input data is not copied

**Returns****df** [DataFrame with DatetimeIndex]**pandas.DataFrame.to\_xarray**`DataFrame.to_xarray()`

Return an xarray object from the pandas object.

**Returns****a DataArray for a Series****a Dataset for a DataFrame****a DataArray for higher dims****Notes**See the [xarray docs](#)**Examples**

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                        'B' : ['foo', 'bar', 'foo'],
                        'C' : np.arange(4.,7)})
>>> df
   A  B    C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index      (index) int64 0 1 2
Data variables:
  A            (index) int64 1 1 2
  B            (index) object 'foo' 'bar' 'foo'
  C            (index) float64 4.0 5.0 6.0
```

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
                        'B' : ['foo', 'bar', 'foo'],
                        'C' : np.arange(4.,7)}
                        ).set_index(['B', 'A'])

>>> df
      C
B  A
foo 1  4.0
bar 1  5.0
foo 2  6.0
```

```
>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B          (B) object 'bar' 'foo'
  * A          (A) int64 1 2
Data variables:
  C            (B, A) float64 5.0 nan 4.0 6.0
```

```
>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                 items=list('ABCD'),
                 major_axis=pd.date_range('20130101', periods=3),
                 minor_axis=['first', 'second'])

>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second
```

```
>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
         [ 2,  3],
         [ 4,  5]],
       [[ 6,  7],
         [ 8,  9],
         [10, 11]],
       [[12, 13],
         [14, 15],
         [16, 17]],
       [[18, 19],
         [20, 21],
         [22, 23]]])
Coordinates:
```

(continues on next page)

(continued from previous page)

```

* items      (items) object 'A' 'B' 'C' 'D'
* major_axis (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03
→ # noqa
* minor_axis (minor_axis) object 'first' 'second'

```

**pandas.DataFrame.transform**

`DataFrame.transform(func, *args, **kwargs)`

Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values

New in version 0.20.0.

**Parameters** `func` : callable, string, dictionary, or list of string/callables

To apply to column

Accepted Combinations are:

- string function name
- function
- list of functions
- dict of column names -> functions (or list of functions)

**Returns**

**transformed** [NDFrame]

**See also:**

`pandas.NDFrame.aggregate`, `pandas.NDFrame.apply`

**Examples**

```

>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'],
...                    index=pd.date_range('1/1/2000', periods=10))
df.iloc[3:7] = np.nan

```

```

>>> df.transform(lambda x: (x - x.mean()) / x.std())

```

	A	B	C
2000-01-01	0.579457	1.236184	0.123424
2000-01-02	0.370357	-0.605875	-1.231325
2000-01-03	1.455756	-0.277446	0.288967
2000-01-04	NaN	NaN	NaN
2000-01-05	NaN	NaN	NaN
2000-01-06	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN
2000-01-08	-0.498658	1.274522	1.642524
2000-01-09	-0.540524	-1.012676	-0.828968
2000-01-10	-1.366388	-0.614710	0.005378

## pandas.DataFrame.transpose

`DataFrame.transpose(*args, **kwargs)`

Transpose index and columns.

Reflect the DataFrame over its main diagonal by writing rows as columns and vice-versa. The property *T* is an accessor to the method `transpose()`.

**Parameters** `copy` : bool, default False

If True, the underlying data is copied. Otherwise (default), no copy is made if possible.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns** `DataFrame`

The transposed DataFrame.

**See also:**

`numpy.transpose` Permute the dimensions of a given array.

## Notes

Transposing a DataFrame with mixed dtypes will result in a homogeneous DataFrame with the *object* dtype. In such a case, a copy of the data is always made.

## Examples

### Square DataFrame with homogeneous dtype

```
>>> d1 = {'col1': [1, 2], 'col2': [3, 4]}
>>> df1 = pd.DataFrame(data=d1)
>>> df1
   col1  col2
0     1     3
1     2     4
```

```
>>> df1_transposed = df1.T # or df1.transpose()
>>> df1_transposed
   0  1
col1 1  2
col2 3  4
```

When the dtype is homogeneous in the original DataFrame, we get a transposed DataFrame with the same dtype:

```
>>> df1.dtypes
col1    int64
col2    int64
dtype: object
>>> df1_transposed.dtypes
```

(continues on next page)



(continued from previous page)

```
0    int64
1    int64
dtype: object
```

**Non-square DataFrame with mixed dtypes**

```
>>> d2 = {'name': ['Alice', 'Bob'],
...       'score': [9.5, 8],
...       'employed': [False, True],
...       'kids': [0, 0]}
>>> df2 = pd.DataFrame(data=d2)
>>> df2
   name  score  employed  kids
0  Alice   9.5     False    0
1   Bob    8.0      True    0
```

```
>>> df2_transposed = df2.T # or df2.transpose()
>>> df2_transposed
      0    1
name   Alice  Bob
score    9.5    8
employed False  True
kids       0    0
```

When the DataFrame has mixed dtypes, we get a transposed DataFrame with the *object* dtype:

```
>>> df2.dtypes
name      object
score    float64
employed    bool
kids      int64
dtype: object
>>> df2_transposed.dtypes
0    object
1    object
dtype: object
```

**pandas.DataFrame.truediv**

`DataFrame.truediv` (*other*, *axis*='columns', *level*=None, *fill\_value*=None)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a *fill\_value* for missing data in one of the inputs.

**Parameters**

**other** [Series, DataFrame, or constant]

**axis** : {0, 1, 'index', 'columns'}

For Series input, axis to match Series index on

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : None or float value, default None

Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

#### Returns

**result** [DataFrame]

#### See also:

`DataFrame.rtruediv`

#### Notes

Mismatched indices will be unioned together

#### Examples

None

### **pandas.DataFrame.truncate**

`DataFrame.truncate` (*before=None, after=None, axis=None, copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

**Parameters** **before** : date, string, int

Truncate all rows before this index value.

**after** : date, string, int

Truncate all rows after this index value.

**axis** : {0 or 'index', 1 or 'columns'}, optional

Axis to truncate. Truncates the index (rows) by default.

**copy** : boolean, default is True,

Return a copy of the truncated section.

**Returns** **type of caller**

The truncated Series or DataFrame.

#### See also:

`DataFrame.loc` Select a subset of a DataFrame by label.

`DataFrame.iloc` Select a subset of a DataFrame by position.

#### Notes

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

## Examples

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                     'B': ['f', 'g', 'h', 'i', 'j'],
...                     'C': ['k', 'l', 'm', 'n', 'o']},
...                     index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
```

(continues on next page)

(continued from previous page)

```
2016-01-09 23:59:59    1
2016-01-10 00:00:00    1
```

Because the index is a `DatetimeIndex` containing only dates, we can specify *before* and *after* as strings. They will be coerced to Timestamps before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
      A
2016-01-09 23:59:56    1
2016-01-09 23:59:57    1
2016-01-09 23:59:58    1
2016-01-09 23:59:59    1
2016-01-10 00:00:00    1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.

```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
      A
2016-01-10 23:59:55    1
2016-01-10 23:59:56    1
2016-01-10 23:59:57    1
2016-01-10 23:59:58    1
2016-01-10 23:59:59    1
```

## pandas.DataFrame.tshift

`DataFrame.tshift` (*periods=1, freq=None, axis=0*)

Shift the time index, using the index's frequency if available.

**Parameters** `periods` : int

Number of periods to move, can be positive or negative

**freq** : `DateOffset`, `timedelta`, or time rule string, default `None`

Increment to use from the `tseries` module or time rule (e.g. 'EOM')

**axis** : int or basestring

Corresponds to the axis that contains the Index

**Returns**

**shifted** [NDFrame]

## Notes

If `freq` is not specified then tries to use the `freq` or `inferred_freq` attributes of the index. If neither of those attributes exist, a `ValueError` is thrown

## pandas.DataFrame.tz\_convert

`DataFrame.tz_convert` (*tz, axis=0, level=None, copy=True*)

Convert tz-aware axis to target time zone.

**Parameters****tz** [string or pytz.timezone object]**axis** [the axis to convert]**level** : int, str, default None

If axis is a MultiIndex, convert a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**Raises** **TypeError**

If the axis is tz-naive.

**pandas.DataFrame.tz\_localize**`DataFrame.tz_localize(tz, axis=0, level=None, copy=True, ambiguous='raise')`

Localize tz-naive TimeSeries to target time zone.

**Parameters****tz** [string or pytz.timezone object]**axis** [the axis to localize]**level** : int, str, default None

If axis is a MultiIndex, localize a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**ambiguous** : 'infer', bool-ndarray, 'NaT', default 'raise'

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

**Raises** **TypeError**

If the TimeSeries is tz-aware and tz is not None.

**pandas.DataFrame.unstack**`DataFrame.unstack(level=-1, fill_value=None)`

Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels. If the index is not a MultiIndex, the output will be a Series (the analogue of stack when the columns are not a MultiIndex). The level involved will automatically get sorted.

**Parameters** **level** : int, string, or list of these, default -1 (last level)

Level(s) of index to unstack, can pass level name

**fill\_value** : replace NaN with this value if the unstack produces missing values  
New in version 0.18.0.

### Returns

**unstacked** [DataFrame or Series]

See also:

**DataFrame.pivot** Pivot a table based on column values.

**DataFrame.stack** Pivot a level of the column labels (inverse operation from *unstack*).

### Examples

```
>>> index = pd.MultiIndex.from_tuples([('one', 'a'), ('one', 'b'),
...                                   ('two', 'a'), ('two', 'b')])
>>> s = pd.Series(np.arange(1.0, 5.0), index=index)
>>> s
one  a    1.0
     b    2.0
two  a    3.0
     b    4.0
dtype: float64
```

```
>>> s.unstack(level=-1)
     a    b
one  1.0  2.0
two  3.0  4.0
```

```
>>> s.unstack(level=0)
     one  two
a    1.0   3.0
b    2.0   4.0
```

```
>>> df = s.unstack(level=0)
>>> df.unstack()
one  a    1.0
     b    2.0
two  a    3.0
     b    4.0
dtype: float64
```

### pandas.DataFrame.update

**DataFrame.update** (*other*, *join*='left', *overwrite*=True, *filter\_func*=None, *raise\_conflict*=False)

Modify in place using non-NA values from another DataFrame.

Aligns on indices. There is no return value.

**Parameters** **other** : DataFrame, or object coercible into a DataFrame

Should have at least one matching index/column label with the original DataFrame. If a Series is passed, its name attribute must be set, and that will be used as the column name to align with the original DataFrame.

**join** : {'left'}, default 'left'

Only left join is implemented, keeping the index and columns of the original object.

**overwrite** : bool, default True

How to handle non-NA values for overlapping keys:

- True: overwrite original DataFrame's values with values from *other*.
- False: only update values that are NA in the original DataFrame.

**filter\_func** : callable(1d-array) -> boolean 1d-array, optional

Can choose to replace values other than NA. Return True for values that should be updated.

**raise\_conflict** : bool, default False

If True, will raise a ValueError if the DataFrame and *other* both contain non-NA data in the same place.

#### Raises ValueError

When *raise\_conflict* is True and there's overlapping non-NA data.

**See also:**

`dict.update` Similar method for dictionaries.

`DataFrame.merge` For column(s)-on-columns(s) operations.

#### Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                    'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, 5, 6],
...                        'C': [7, 8, 9]})
>>> df.update(new_df)
>>> df
   A  B
0  1  4
1  2  5
2  3  6
```

The DataFrame's length does not increase as a result of the update, only values at matching index/column labels are updated.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                    'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e', 'f', 'g', 'h', 'i']})
>>> df.update(new_df)
>>> df
   A  B
0  a  d
```

(continues on next page)

(continued from previous page)

```
1  b  e
2  c  f
```

For Series, it's name attribute must be set.

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                     'B': ['x', 'y', 'z']})
>>> new_column = pd.Series(['d', 'e'], name='B', index=[0, 2])
>>> df.update(new_column)
>>> df
   A  B
0  a  d
1  b  y
2  c  e
>>> df = pd.DataFrame({'A': ['a', 'b', 'c'],
...                     'B': ['x', 'y', 'z']})
>>> new_df = pd.DataFrame({'B': ['d', 'e']}, index=[1, 2])
>>> df.update(new_df)
>>> df
   A  B
0  a  x
1  b  d
2  c  e
```

If *other* contains NaNs the corresponding values are not updated in the original dataframe.

```
>>> df = pd.DataFrame({'A': [1, 2, 3],
...                     'B': [400, 500, 600]})
>>> new_df = pd.DataFrame({'B': [4, np.nan, 6]})
>>> df.update(new_df)
>>> df
   A    B
0  1  4.0
1  2 500.0
2  3  6.0
```

## pandas.DataFrame.var

`DataFrame.var` (*axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the `ddof` argument

### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**ddof** : int, default 1



Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**var** [Series or DataFrame (if level specified)]

### pandas.DataFrame.where

`DataFrame.where(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)`

Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

**Parameters** **cond** : boolean NDFrame, array-like, or callable

Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

**other** : scalar, NDFrame, or callable

Entries where *cond* is False are replaced with corresponding value from *other*. If *other* is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

**axis** [alignment axis if needed, default None]

**level** [alignment level if needed, default None]

**errors** : str, {'raise', 'ignore'}, default 'raise'

- **raise** : allow exceptions to be raised
- **ignore** : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

### Returns

**wh** [same type as caller]

### See also:

`DataFrame.mask()`

### Notes

The `where` method is an application of the if-then idiom. For each element in the calling `DataFrame`, if `cond` is `True` the element is used; otherwise the corresponding element from the `DataFrame` `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in [indexing](#).

### Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A  B
0  True True
```

(continues on next page)

(continued from previous page)

```

1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
      A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True

```

**pandas.DataFrame.xs****DataFrame.xs** (*key*, *axis=0*, *level=None*, *drop\_level=True*)

Returns a cross-section (row(s) or column(s)) from the Series/DataFrame. Defaults to cross-section on the rows (*axis=0*).

**Parameters** **key** : object

Some label contained in the index, or partially in a MultiIndex

**axis** : int, default 0

Axis to retrieve cross-section on

**level** : object, defaults to first n levels (*n=1* or *len(key)*)

In case of a key partially contained in a MultiIndex, indicate which levels are used. Levels can be referred by label or position.

**drop\_level** : boolean, default True

If False, returns object with same levels as self.

**Returns****xs** [Series or DataFrame]**Notes**

*xs* is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels. It is a superset of *xs* functionality, see [MultiIndex Slicers](#)

**Examples**

```

>>> df
      A  B  C
a    4  5  2
b    4  0  9
c    9  7  3
>>> df.xs('a')
      A
a    4

```

(continues on next page)

(continued from previous page)

```

B      5
C      2
Name: a
>>> df.xs('C', axis=1)
a      2
b      9
c      3
Name: C

```

```

>>> df
      first second third   A  B  C  D
bar    one     1     4  1  8  9
      two     1     7  5  5  0
baz    one     1     6  6  8  0
      three    2     5  3  5  3
>>> df.xs(('baz', 'three'))
      A  B  C  D
third
2      5  3  5  3
>>> df.xs('one', level=1)
      A  B  C  D
first third
bar    1     4  1  8  9
baz    1     6  6  8  0
>>> df.xs(('baz', 2), level=[0, 'third'])
      A  B  C  D
second
three  5  3  5  3

```

### 34.4.2 Attributes and underlying data

#### Axes

<code>DataFrame.index</code>	The index (row labels) of the DataFrame.
<code>DataFrame.columns</code>	The column labels of the DataFrame.
<code>DataFrame.dtypes</code>	Return the dtypes in the DataFrame.
<code>DataFrame.ftypes</code>	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
<code>DataFrame.get_dtype_counts()</code>	Return counts of unique dtypes in this object.
<code>DataFrame.get_ftype_counts()</code>	(DEPRECATED) Return counts of unique ftypes in this object.
<code>DataFrame.select_dtypes([include, exclude])</code>	Return a subset of the DataFrame's columns based on the column dtypes.
<code>DataFrame.values</code>	Return a Numpy representation of the DataFrame.
<code>DataFrame.get_values()</code>	Return an ndarray after converting sparse values to dense.
<code>DataFrame.axes</code>	Return a list representing the axes of the DataFrame.
<code>DataFrame.ndim</code>	Return an int representing the number of axes / array dimensions.

Continued on next page

Table 63 – continued from previous page

<code>DataFrame.size</code>	Return an int representing the number of elements in this object.
<code>DataFrame.shape</code>	Return a tuple representing the dimensionality of the DataFrame.
<code>DataFrame.memory_usage([index, deep])</code>	Return the memory usage of each column in bytes.
<code>DataFrame.empty</code>	Indicator whether DataFrame is empty.
<code>DataFrame.is_copy</code>	

#### 34.4.2.1 pandas.DataFrame.is\_copy

`DataFrame.is_copy`

### 34.4.3 Conversion

<code>DataFrame.astype(dtype[, copy, errors])</code>	Cast a pandas object to a specified dtype <code>dtype</code> .
<code>DataFrame.convert_objects([convert_dates, ...])</code>	(DEPRECATED) Attempt to infer better dtype for object columns.
<code>DataFrame.infer_objects()</code>	Attempt to infer better dtypes for object columns.
<code>DataFrame.copy([deep])</code>	Make a copy of this object's indices and data.
<code>DataFrame.isna()</code>	Detect missing values.
<code>DataFrame.notna()</code>	Detect existing (non-missing) values.
<code>DataFrame.bool()</code>	Return the bool of a single element PandasObject.

### 34.4.4 Indexing, iteration

<code>DataFrame.head([n])</code>	Return the first <i>n</i> rows.
<code>DataFrame.at</code>	Access a single value for a row/column label pair.
<code>DataFrame.iat</code>	Access a single value for a row/column pair by integer position.
<code>DataFrame.loc</code>	Access a group of rows and columns by label(s) or a boolean array.
<code>DataFrame.iloc</code>	Purely integer-location based indexing for selection by position.
<code>DataFrame.insert(loc, column, value[, ...])</code>	Insert column into DataFrame at specified location.
<code>DataFrame.insert(loc, column, value[, ...])</code>	Insert column into DataFrame at specified location.
<code>DataFrame.__iter__()</code>	Iterate over infor axis
<code>DataFrame.items()</code>	Iterator over (column name, Series) pairs.
<code>DataFrame.keys()</code>	Get the 'info axis' (see Indexing for more)
<code>DataFrame.iteritems()</code>	Iterator over (column name, Series) pairs.
<code>DataFrame.iterrows()</code>	Iterate over DataFrame rows as (index, Series) pairs.
<code>DataFrame.itertuples([index, name])</code>	Iterate over DataFrame rows as namedtuples, with index value as first element of the tuple.
<code>DataFrame.lookup(row_labels, col_labels)</code>	Label-based "fancy indexing" function for DataFrame.
<code>DataFrame.pop(item)</code>	Return item and drop from frame.
<code>DataFrame.tail([n])</code>	Return the last <i>n</i> rows.
<code>DataFrame.xs(key[, axis, level, drop_level])</code>	Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

Continued on next page

Table 65 – continued from previous page

<code>DataFrame.get(key[, default])</code>	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>DataFrame.isin(values)</code>	Return boolean DataFrame showing whether each element in the DataFrame is contained in values.
<code>DataFrame.where(cond[, other, inplace, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .
<code>DataFrame.mask(cond[, other, inplace, axis, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>DataFrame.query(expr[, inplace])</code>	Query the columns of a frame with a boolean expression.

#### 34.4.4.1 pandas.DataFrame.\_\_iter\_\_

`DataFrame.__iter__()`  
Iterate over infor axis

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the [indexing documentation](#).

#### 34.4.5 Binary operator functions

<code>DataFrame.add(other[, axis, level, fill_value])</code>	Addition of dataframe and other, element-wise (binary operator <i>add</i> ).
<code>DataFrame.sub(other[, axis, level, fill_value])</code>	Subtraction of dataframe and other, element-wise (binary operator <i>sub</i> ).
<code>DataFrame.mul(other[, axis, level, fill_value])</code>	Multiplication of dataframe and other, element-wise (binary operator <i>mul</i> ).
<code>DataFrame.div(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>DataFrame.truediv(other[, axis, level, ...])</code>	Floating division of dataframe and other, element-wise (binary operator <i>truediv</i> ).
<code>DataFrame.floordiv(other[, axis, level, ...])</code>	Integer division of dataframe and other, element-wise (binary operator <i>floordiv</i> ).
<code>DataFrame.mod(other[, axis, level, fill_value])</code>	Modulo of dataframe and other, element-wise (binary operator <i>mod</i> ).
<code>DataFrame.pow(other[, axis, level, fill_value])</code>	Exponential power of dataframe and other, element-wise (binary operator <i>pow</i> ).
<code>DataFrame.dot(other)</code>	Matrix multiplication with DataFrame or Series objects.
<code>DataFrame.radd(other[, axis, level, fill_value])</code>	Addition of dataframe and other, element-wise (binary operator <i>radd</i> ).
<code>DataFrame.rsub(other[, axis, level, fill_value])</code>	Subtraction of dataframe and other, element-wise (binary operator <i>rsub</i> ).
<code>DataFrame.rmul(other[, axis, level, fill_value])</code>	Multiplication of dataframe and other, element-wise (binary operator <i>rmul</i> ).
<code>DataFrame.rdiv(other[, axis, level, fill_value])</code>	Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).
<code>DataFrame.rtruediv(other[, axis, level, ...])</code>	Floating division of dataframe and other, element-wise (binary operator <i>rtruediv</i> ).

Continued on next page

Table 66 – continued from previous page

<code>DataFrame.rfloordiv(other[, axis, level, ...])</code>	Integer division of dataframe and other, element-wise (binary operator <i>rfloordiv</i> ).
<code>DataFrame.rmod(other[, axis, level, fill_value])</code>	Modulo of dataframe and other, element-wise (binary operator <i>rmod</i> ).
<code>DataFrame.rpow(other[, axis, level, fill_value])</code>	Exponential power of dataframe and other, element-wise (binary operator <i>rpow</i> ).
<code>DataFrame.lt(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>lt</i>
<code>DataFrame.gt(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>gt</i>
<code>DataFrame.le(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>le</i>
<code>DataFrame.ge(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>ge</i>
<code>DataFrame.ne(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>ne</i>
<code>DataFrame.eq(other[, axis, level])</code>	Wrapper for flexible comparison methods <i>eq</i>
<code>DataFrame.combine(other, func[, fill_value, ...])</code>	Add two DataFrame objects and do not propagate NaN values, so if for a (column, time) one frame is missing a value, it will default to the other frame's value (which might be NaN as well)
<code>DataFrame.combine_first(other)</code>	Combine two DataFrame objects and default to non-null values in frame calling the method.

### 34.4.6 Function application, GroupBy & Window

<code>DataFrame.apply(func[, axis, broadcast, ...])</code>	Apply a function along an axis of the DataFrame.
<code>DataFrame.applymap(func)</code>	Apply a function to a Dataframe elementwise.
<code>DataFrame.pipe(func, *args, **kwargs)</code>	Apply <code>func(self, *args, **kwargs)</code>
<code>DataFrame.agg(func[, axis])</code>	Aggregate using one or more operations over the specified axis.
<code>DataFrame.aggregate(func[, axis])</code>	Aggregate using one or more operations over the specified axis.
<code>DataFrame.transform(func, *args, **kwargs)</code>	Call function producing a like-indexed NDFrame and return a NDFrame with the transformed values
<code>DataFrame.groupby([by, axis, level, ...])</code>	Group series using mapper (dict or key function, apply given function to group, return result as series) or by a series of columns.
<code>DataFrame.rolling(window[, min_periods, ...])</code>	Provides rolling window calculations.
<code>DataFrame.expanding([min_periods, center, axis])</code>	Provides expanding transformations.
<code>DataFrame.ewm([com, span, halflife, alpha, ...])</code>	Provides exponential weighted functions

### 34.4.7 Computations / Descriptive Stats

<code>DataFrame.abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>DataFrame.all([axis, bool_only, skipna, level])</code>	Return whether all elements are True, potentially over an axis.
<code>DataFrame.any([axis, bool_only, skipna, level])</code>	Return whether any element is True over requested axis.
<code>DataFrame.clip([lower, upper, axis, inplace])</code>	Trim values at input threshold(s).
<code>DataFrame.clip_lower(threshold[, axis, inplace])</code>	Return copy of the input with values below a threshold truncated.

Continued on next page

Table 68 – continued from previous page

<code>DataFrame.clip_upper(threshold[, axis, inplace])</code>	Return copy of input with values above given value(s) truncated.
<code>DataFrame.compound([axis, skipna, level])</code>	Return the compound percentage of the values for the requested axis
<code>DataFrame.corr([method, min_periods])</code>	Compute pairwise correlation of columns, excluding NA/null values
<code>DataFrame.corrwith(other[, axis, drop])</code>	Compute pairwise correlation between rows or columns of two DataFrame objects.
<code>DataFrame.count([axis, level, numeric_only])</code>	Count non-NA cells for each column or row.
<code>DataFrame.cov([min_periods])</code>	Compute pairwise covariance of columns, excluding NA/null values.
<code>DataFrame.cummax([axis, skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>DataFrame.cummin([axis, skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>DataFrame.cumprod([axis, skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>DataFrame.cumsum([axis, skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>DataFrame.describe([percentiles, include, ...])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>DataFrame.diff([periods, axis])</code>	First discrete difference of element.
<code>DataFrame.eval(expr[, inplace])</code>	Evaluate a string describing operations on DataFrame columns.
<code>DataFrame.kurt([axis, skipna, level, ...])</code>	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>DataFrame.kurtosis([axis, skipna, level, ...])</code>	Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0).
<code>DataFrame.mad([axis, skipna, level])</code>	Return the mean absolute deviation of the values for the requested axis
<code>DataFrame.max([axis, skipna, level, ...])</code>	This method returns the maximum of the values in the object.
<code>DataFrame.mean([axis, skipna, level, ...])</code>	Return the mean of the values for the requested axis
<code>DataFrame.median([axis, skipna, level, ...])</code>	Return the median of the values for the requested axis
<code>DataFrame.min([axis, skipna, level, ...])</code>	This method returns the minimum of the values in the object.
<code>DataFrame.mode([axis, numeric_only])</code>	Gets the mode(s) of each element along the axis selected.
<code>DataFrame.pct_change([periods, fill_method, ...])</code>	Percentage change between the current and a prior element.
<code>DataFrame.prod([axis, skipna, level, ...])</code>	Return the product of the values for the requested axis
<code>DataFrame.product([axis, skipna, level, ...])</code>	Return the product of the values for the requested axis
<code>DataFrame.quantile([q, axis, numeric_only, ...])</code>	Return values at the given quantile over requested axis, a la numpy.percentile.
<code>DataFrame.rank([axis, method, numeric_only, ...])</code>	Compute numerical data ranks (1 through n) along axis.
<code>DataFrame.round([decimals])</code>	Round a DataFrame to a variable number of decimal places.

Continued on next page



Table 68 – continued from previous page

<code>DataFrame.sem([axis, skipna, level, ddof, ...])</code>	Return unbiased standard error of the mean over requested axis.
<code>DataFrame.skew([axis, skipna, level, ...])</code>	Return unbiased skew over requested axis Normalized by N-1
<code>DataFrame.sum([axis, skipna, level, ...])</code>	Return the sum of the values for the requested axis
<code>DataFrame.std([axis, skipna, level, ddof, ...])</code>	Return sample standard deviation over requested axis.
<code>DataFrame.var([axis, skipna, level, ddof, ...])</code>	Return unbiased variance over requested axis.
<code>DataFrame.nunique([axis, dropna])</code>	Return Series with number of distinct observations over requested axis.

### 34.4.8 Reindexing / Selection / Label manipulation

<code>DataFrame.add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>DataFrame.add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>DataFrame.align(other[, join, axis, level, ...])</code>	Align two objects on their axes with the specified join method for each axis Index
<code>DataFrame.at_time(time[, asof])</code>	Select values at particular time of day (e.g.
<code>DataFrame.between_time(start_time, end_time)</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>DataFrame.drop([labels, axis, index, ...])</code>	Drop specified labels from rows or columns.
<code>DataFrame.drop_duplicates([subset, keep, ...])</code>	Return DataFrame with duplicate rows removed, optionally only considering certain columns
<code>DataFrame.duplicated([subset, keep])</code>	Return boolean Series denoting duplicate rows, optionally only considering certain columns
<code>DataFrame.equals(other)</code>	Determines if two NDFrame objects contain the same elements.
<code>DataFrame.filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>DataFrame.first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>DataFrame.head([n])</code>	Return the first <i>n</i> rows.
<code>DataFrame.idxmax([axis, skipna])</code>	Return index of first occurrence of maximum over requested axis.
<code>DataFrame.idxmin([axis, skipna])</code>	Return index of first occurrence of minimum over requested axis.
<code>DataFrame.last(offset)</code>	Convenience method for subsetting final periods of time series data based on a date offset.
<code>DataFrame.reindex([labels, index, columns, ...])</code>	Conform DataFrame to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>DataFrame.reindex_axis(labels[, axis, ...])</code>	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>DataFrame.reindex_like(other[, method, ...])</code>	Return an object with matching indices to myself.
<code>DataFrame.rename([mapper, index, columns, ...])</code>	Alter axes labels.
<code>DataFrame.rename_axis(mapper[, axis, copy, ...])</code>	Alter the name of the index or columns.

Continued on next page

Table 69 – continued from previous page

<code>DataFrame.reset_index([level, drop, ...])</code>	For DataFrame with multi-level index, return new DataFrame with labeling information in the columns under the index names, defaulting to 'level_0', 'level_1', etc.
<code>DataFrame.sample([n, frac, replace, ...])</code>	Return a random sample of items from an axis of object.
<code>DataFrame.select(crit[, axis])</code>	(DEPRECATED) Return data corresponding to axis labels matching criteria
<code>DataFrame.set_axis(labels[, axis, inplace])</code>	Assign desired index to given axis.
<code>DataFrame.set_index(keys[, drop, append, ...])</code>	Set the DataFrame index (row labels) using one or more existing columns.
<code>DataFrame.tail([n])</code>	Return the last <i>n</i> rows.
<code>DataFrame.take(indices[, axis, convert, is_copy])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>DataFrame.truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.

### 34.4.9 Missing data handling

<code>DataFrame.dropna([axis, how, thresh, ...])</code>	Remove missing values.
<code>DataFrame.fillna([value, method, axis, ...])</code>	Fill NA/NaN values using the specified method
<code>DataFrame.replace([to_replace, value, ...])</code>	Replace values given in <i>to_replace</i> with <i>value</i> .
<code>DataFrame.interpolate([method, axis, limit, ...])</code>	Interpolate values according to different methods.

### 34.4.10 Reshaping, sorting, transposing

<code>DataFrame.pivot([index, columns, values])</code>	Return reshaped DataFrame organized by given index / column values.
<code>DataFrame.pivot_table([values, index, ...])</code>	Create a spreadsheet-style pivot table as a DataFrame.
<code>DataFrame.reorder_levels(order[, axis])</code>	Rearrange index levels using input order.
<code>DataFrame.sort_values(by[, axis, ascending, ...])</code>	Sort by the values along either axis
<code>DataFrame.sort_index([axis, level, ...])</code>	Sort object by labels (along an axis)
<code>DataFrame.nlargest(n, columns[, keep])</code>	Return the first <i>n</i> rows ordered by <i>columns</i> in descending order.
<code>DataFrame.nsmallest(n, columns[, keep])</code>	Get the rows of a DataFrame sorted by the <i>n</i> smallest values of <i>columns</i> .
<code>DataFrame.swaplevel([i, j, axis])</code>	Swap levels <i>i</i> and <i>j</i> in a MultiIndex on a particular axis
<code>DataFrame.stack([level, dropna])</code>	Stack the prescribed level(s) from columns to index.
<code>DataFrame.unstack([level, fill_value])</code>	Pivot a level of the (necessarily hierarchical) index labels, returning a DataFrame having a new level of column labels whose inner-most level consists of the pivoted index labels.
<code>DataFrame.swapaxes(axis1, axis2[, copy])</code>	Interchange axes and swap values axes appropriately
<code>DataFrame.melt([id_vars, value_vars, ...])</code>	“Unpivots” a DataFrame from wide format to long format, optionally leaving identifier variables set.
<code>DataFrame.squeeze([axis])</code>	Squeeze length 1 dimensions.
<code>DataFrame.to_panel()</code>	(DEPRECATED) Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.

Continued on next page

Table 71 – continued from previous page

<code>DataFrame.to_xarray()</code>	Return an xarray object from the pandas object.
<code>DataFrame.T</code>	Transpose index and columns.
<code>DataFrame.transpose(*args, **kwargs)</code>	Transpose index and columns.

### 34.4.11 Combining / joining / merging

<code>DataFrame.append(other[, ignore_index, ...])</code>	Append rows of <i>other</i> to the end of this frame, returning a new object.
<code>DataFrame.assign(**kwargs)</code>	Assign new columns to a DataFrame, returning a new object (a copy) with the new columns added to the original ones.
<code>DataFrame.join(other[, on, how, lsuffix, ...])</code>	Join columns with other DataFrame either on index or on a key column.
<code>DataFrame.merge(right[, how, on, left_on, ...])</code>	Merge DataFrame objects by performing a database-style join operation by columns or indexes.
<code>DataFrame.update(other[, join, overwrite, ...])</code>	Modify in place using non-NA values from another DataFrame.

### 34.4.12 Time series-related

<code>DataFrame.asfreq(freq[, method, how, ...])</code>	Convert TimeSeries to specified frequency.
<code>DataFrame.asof(where[, subset])</code>	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)
<code>DataFrame.shift([periods, freq, axis])</code>	Shift index by desired number of periods with an optional time freq
<code>DataFrame.slice_shift([periods, axis])</code>	Equivalent to <i>shift</i> without copying data.
<code>DataFrame.tshift([periods, freq, axis])</code>	Shift the time index, using the index's frequency if available.
<code>DataFrame.first_valid_index()</code>	Return index for first non-NA/null value.
<code>DataFrame.last_valid_index()</code>	Return index for last non-NA/null value.
<code>DataFrame.resample(rule[, how, axis, ...])</code>	Convenience method for frequency conversion and resampling of time series.
<code>DataFrame.to_period([freq, axis, copy])</code>	Convert DataFrame from DatetimeIndex to PeriodIndex with desired frequency (inferred from index if not passed)
<code>DataFrame.to_timestamp([freq, how, axis, copy])</code>	Cast to DatetimeIndex of timestamps, at <i>beginning</i> of period
<code>DataFrame.tz_convert(tz[, axis, level, copy])</code>	Convert tz-aware axis to target time zone.
<code>DataFrame.tz_localize(tz[, axis, level, ...])</code>	Localize tz-naive TimeSeries to target time zone.

### 34.4.13 Plotting

`DataFrame.plot` is both a callable method and a namespace attribute for specific plotting methods of the form `DataFrame.plot.<kind>`.

<code>DataFrame.plot([x, y, kind, ax, ...])</code>	DataFrame plotting accessor and method
--	--

<code>DataFrame.plot.area(x, y)</code>	Area plot
<code>DataFrame.plot.bar(x, y)</code>	Vertical bar plot.
<code>DataFrame.plot.barh(x, y)</code>	Make a horizontal bar plot.
<code>DataFrame.plot.box([by])</code>	Make a box plot of the DataFrame columns.
<code>DataFrame.plot.density([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>DataFrame.plot.hexbin(x, y[, C, ...])</code>	Generate a hexagonal binning plot.
<code>DataFrame.plot.hist([by, bins])</code>	Draw one histogram of the DataFrame's columns.
<code>DataFrame.plot.kde([bw_method, ind])</code>	Generate Kernel Density Estimate plot using Gaussian kernels.
<code>DataFrame.plot.line(x, y)</code>	Plot DataFrame columns as lines.
<code>DataFrame.plot.pie([y])</code>	Generate a pie plot.
<code>DataFrame.plot.scatter(x, y[, s, c])</code>	Create a scatter plot with varying marker point size and color.

#### 34.4.13.1 pandas.DataFrame.plot.area

`DataFrame.plot.area` (*x=None, y=None, \*\*kws*)

Area plot

**Parameters** *x, y* : label or position, optional

Coordinates for each point.

**\*\*kws** : optional

Additional keyword arguments are documented in `pandas.DataFrame.plot()`.

**Returns**

**axes** [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

#### 34.4.13.2 pandas.DataFrame.plot.bar

`DataFrame.plot.bar` (*x=None, y=None, \*\*kws*)

Vertical bar plot.

A bar plot is a plot that presents categorical data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters** *x* : label or position, optional

Allows plotting of one column versus another. If not specified, the index of the DataFrame is used.

*y* : label or position, optional

Allows plotting of one column versus another. If not specified, all numerical columns are used.

**\*\*kws**

Additional keyword arguments are documented in `pandas.DataFrame.plot()`.

**Returns** **axes** : `matplotlib.axes.Axes` or `np.ndarray` of them

An ndarray is returned with one `matplotlib.axes.Axes` per column when `subplots=True`.

See also:

`pandas.DataFrame.plot.barh` Horizontal bar plot.

`pandas.DataFrame.plot` Make plots of a DataFrame.

`matplotlib.pyplot.bar` Make a bar plot with matplotlib.

## Examples

Basic plot.

```
>>> df = pd.DataFrame({'lab':['A', 'B', 'C'], 'val':[10, 30, 20]})
>>> ax = df.plot.bar(x='lab', y='val', rot=0)
```

Plot a whole dataframe to a bar plot. Each column is assigned a distinct color, and each row is nested in a group along the horizontal axis.

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> ax = df.plot.bar(rot=0)
```

Instead of nesting, the figure can be split by column with `subplots=True`. In this case, a `numpy.ndarray` of `matplotlib.axes.Axes` are returned.

```
>>> axes = df.plot.bar(rot=0, subplots=True)
>>> axes[1].legend(loc=2)
```

Plot a single column.

```
>>> ax = df.plot.bar(y='speed', rot=0)
```

Plot only selected categories for the DataFrame.

```
>>> ax = df.plot.bar(x='lifespan', rot=0)
```

### 34.4.13.3 pandas.DataFrame.plot.barh

`DataFrame.plot.barh` (`x=None`, `y=None`, `**kws`)

Make a horizontal bar plot.

A horizontal bar plot is a plot that presents quantitative data with rectangular bars with lengths proportional to the values that they represent. A bar plot shows comparisons among discrete categories. One axis of the plot shows the specific categories being compared, and the other axis represents a measured value.

**Parameters** `x` : label or position, default `DataFrame.index`

Column to be used for categories.

`y` : label or position, default All numeric columns in dataframe

Columns to be plotted from the DataFrame.

**\*\*kwds**

Keyword arguments to pass on to `pandas.DataFrame.plot()`.

### Returns

**axes** [`matplotlib.axes.Axes` or `numpy.ndarray` of them.]

See also:

**`pandas.DataFrame.plot.bar`** Vertical bar plot.

**`pandas.DataFrame.plot`** Make plots of DataFrame using matplotlib.

**`matplotlib.axes.Axes.bar`** Plot a vertical bar plot using matplotlib.

## Examples

Basic example

```
>>> df = pd.DataFrame({'lab': ['A', 'B', 'C'], 'val': [10, 30, 20]})
>>> ax = df.plot.barh(x='lab', y='val')
```

Plot a whole DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh()
```

Plot a column of the DataFrame to a horizontal bar plot

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(y='speed')
```

Plot DataFrame versus the desired column

```
>>> speed = [0.1, 17.5, 40, 48, 52, 69, 88]
>>> lifespan = [2, 8, 70, 1.5, 25, 12, 28]
>>> index = ['snail', 'pig', 'elephant',
...         'rabbit', 'giraffe', 'coyote', 'horse']
>>> df = pd.DataFrame({'speed': speed,
...                   'lifespan': lifespan}, index=index)
>>> ax = df.plot.barh(x='lifespan')
```

### 34.4.13.4 pandas.DataFrame.plot.box

`DataFrame.plot.box` (*by=None, \*\*kws*)

Make a box plot of the DataFrame columns.

A box plot is a method for graphically depicting groups of numerical data through their quartiles. The box extends from the Q1 to Q3 quartile values of the data, with a line at the median (Q2). The whiskers extend from the edges of box to show the range of the data. The position of the whiskers is set by default to  $1.5 \times \text{IQR}$  ( $\text{IQR} = Q3 - Q1$ ) from the edges of the box. Outlier points are those past the end of the whiskers.

For further details see Wikipedia's entry for [boxplot](#).

A consideration when using this chart is that the box and the whiskers can overlap, which is very common when plotting small sets of data.

**Parameters** *by* : string or sequence

Column in the DataFrame to group by.

**\*\*kws** : optional

Additional keywords are documented in `pandas.DataFrame.plot()`.

**Returns**

**axes** [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

See also:

`pandas.DataFrame.boxplot` Another method to draw a box plot.

`pandas.Series.plot.box` Draw a box plot from a Series object.

`matplotlib.pyplot.boxplot` Draw a box plot in matplotlib.

### Examples

Draw a box plot from a DataFrame with four columns of randomly generated data.

```
>>> data = np.random.randn(25, 4)
>>> df = pd.DataFrame(data, columns=list('ABCD'))
>>> ax = df.plot.box()
```

### 34.4.13.5 pandas.DataFrame.plot.density

`DataFrame.plot.density` (*bw\_method=None, ind=None, \*\*kws*)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, [kernel density estimation](#) (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters** *bw\_method* : str, scalar or callable, optional

The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.

**ind** : NumPy array or integer, optional

Evaluation points for the estimated PDF. If `None` (default), 1000 equally spaced points are used. If `ind` is a NumPy array, the KDE is evaluated at the points passed. If `ind` is an integer, `ind` number of equally spaced points are used.

**\*\*kwds** : optional

Additional keyword arguments are documented in `pandas.DataFrame.plot()`.

### Returns

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

**See also:**

**scipy.stats.gaussian\_kde** Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

**Series.plot.kde** Generate a KDE plot for a Series.

### Examples

Given several Series of points randomly sampled from unknown distributions, estimate their PDFs using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to overfitting, while using a large bandwidth value may result in underfitting:

```
>>> ax = df.plot.kde(bw_method=0.3)
```

```
>>> ax = df.plot.kde(bw_method=3)
```

Finally, the `ind` parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])
```

#### 34.4.13.6 pandas.DataFrame.plot.hexbin

`DataFrame.plot.hexbin(x, y, C=None, reduce_C_function=None, gridsize=None, **kwds)`

Generate a hexagonal binning plot.

Generate a hexagonal binning plot of `x` versus `y`. If `C` is `None` (the default), this is a histogram of the number of occurrences of the observations at `(x[i], y[i])`.

If `C` is specified, specifies values at given coordinates `(x[i], y[i])`. These values are accumulated for each hexagonal bin and then reduced according to `reduce_C_function`, having as default the NumPy's mean function (`numpy.mean()`). (If `C` is specified, it must also be a 1-D sequence of the same length as `x` and `y`, or a column label.)

**Parameters** `x` : int or str



The column label or position for x points.

**y** : int or str

The column label or position for y points.

**C** : int or str, optional

The column label or position for the value of (x, y) point.

**reduce\_C\_function** : callable, default *np.mean*

Function of one argument that reduces all the values in a bin to a single number (e.g. *np.mean*, *np.max*, *np.sum*, *np.std*).

**gridsize** : int or tuple of (int, int), default 100

The number of hexagons in the x-direction. The corresponding number of hexagons in the y-direction is chosen in a way that the hexagons are approximately regular. Alternatively, gridsize can be a tuple with two elements specifying the number of hexagons in the x-direction and the y-direction.

**\*\*kwds**

Additional keyword arguments are documented in *pandas.DataFrame.plot()*.

**Returns** *matplotlib.AxesSubplot*

The matplotlib *Axes* on which the hexbin is plotted.

**See also:**

*DataFrame.plot* Make plots of a DataFrame.

*matplotlib.pyplot.hexbin* hexagonal binning plot using matplotlib, the matplotlib function that is used under the hood.

## Examples

The following examples are generated with random data from a normal distribution.

```
>>> n = 10000
>>> df = pd.DataFrame({'x': np.random.randn(n),
...                    'y': np.random.randn(n)})
>>> ax = df.plot.hexbin(x='x', y='y', gridsize=20)
```

The next example uses *C* and *np.sum* as *reduce\_C\_function*. Note that 'observations' values ranges from 1 to 5 but the result plot shows values up to more than 25. This is because of the *reduce\_C\_function*.

```
>>> n = 500
>>> df = pd.DataFrame({
...     'coord_x': np.random.uniform(-3, 3, size=n),
...     'coord_y': np.random.uniform(30, 50, size=n),
...     'observations': np.random.randint(1, 5, size=n)
... })
>>> ax = df.plot.hexbin(x='coord_x',
...                     y='coord_y',
...                     C='observations',
...                     reduce_C_function=np.sum,
```

(continues on next page)

(continued from previous page)

```
...         gridsize=10,
...         cmap="viridis")
```

### 34.4.13.7 pandas.DataFrame.plot.hist

`DataFrame.plot.hist` (*by=None, bins=10, \*\*kws*)

Draw one histogram of the DataFrame's columns.

A histogram is a representation of the distribution of data. This function groups the values of all given Series in the DataFrame into bins and draws all bins in one `matplotlib.axes.Axes`. This is useful when the DataFrame's Series are in a similar scale.

**Parameters** *by* : str or sequence, optional

Column in the DataFrame to group by.

*bins* : int, default 10

Number of histogram bins to be used.

**\*\*kws**

Additional keyword arguments are documented in `pandas.DataFrame.plot()`.

**Returns**

*axes* [matplotlib.AxesSubplot histogram.]

See also:

`DataFrame.hist` Draw histograms per DataFrame's Series.

`Series.hist` Draw a histogram with Series' data.

### Examples

When we draw a dice 6000 times, we expect to get each value around 1000 times. But when we draw two dices and sum the result, the distribution is going to be quite different. A histogram illustrates those distributions.

```
>>> df = pd.DataFrame(
...     np.random.randint(1, 7, 6000),
...     columns = ['one'])
>>> df['two'] = df['one'] + np.random.randint(1, 7, 6000)
>>> ax = df.plot.hist(bins=12, alpha=0.5)
```

### 34.4.13.8 pandas.DataFrame.plot.kde

`DataFrame.plot.kde` (*bw\_method=None, ind=None, \*\*kws*)

Generate Kernel Density Estimate plot using Gaussian kernels.

In statistics, [kernel density estimation](#) (KDE) is a non-parametric way to estimate the probability density function (PDF) of a random variable. This function uses Gaussian kernels and includes automatic bandwidth determination.

**Parameters** *bw\_method* : str, scalar or callable, optional

The method used to calculate the estimator bandwidth. This can be 'scott', 'silverman', a scalar constant or a callable. If None (default), 'scott' is used. See `scipy.stats.gaussian_kde` for more information.

**ind** : NumPy array or integer, optional

Evaluation points for the estimated PDF. If None (default), 1000 equally spaced points are used. If *ind* is a NumPy array, the KDE is evaluated at the points passed. If *ind* is an integer, *ind* number of equally spaced points are used.

**\*\*kwargs** : optional

Additional keyword arguments are documented in `pandas.DataFrame.plot()`.

### Returns

**axes** [matplotlib.axes.Axes or numpy.ndarray of them]

See also:

`scipy.stats.gaussian_kde` Representation of a kernel-density estimate using Gaussian kernels. This is the function used internally to estimate the PDF.

`Series.plot.kde` Generate a KDE plot for a Series.

### Examples

Given several Series of points randomly sampled from unknown distributions, estimate their PDFs using KDE with automatic bandwidth determination and plot the results, evaluating them at 1000 equally spaced points (default):

```
>>> df = pd.DataFrame({
...     'x': [1, 2, 2.5, 3, 3.5, 4, 5],
...     'y': [4, 4, 4.5, 5, 5.5, 6, 6],
... })
>>> ax = df.plot.kde()
```

A scalar bandwidth can be specified. Using a small bandwidth value can lead to overfitting, while using a large bandwidth value may result in underfitting:

```
>>> ax = df.plot.kde(bw_method=0.3)
```

```
>>> ax = df.plot.kde(bw_method=3)
```

Finally, the *ind* parameter determines the evaluation points for the plot of the estimated PDF:

```
>>> ax = df.plot.kde(ind=[1, 2, 3, 4, 5, 6])
```

#### 34.4.13.9 pandas.DataFrame.plot.line

`DataFrame.plot.line` (*x=None, y=None, \*\*kwargs*)

Plot DataFrame columns as lines.

This function is useful to plot lines using DataFrame's values as coordinates.

**Parameters** *x* : int or str, optional

Columns to use for the horizontal axis. Either the location or the label of the columns to be used. By default, it will use the DataFrame indices.

**y** : int, str, or list of them, optional

The values to be plotted. Either the location or the label of the columns to be used. By default, it will use the remaining DataFrame numeric columns.

**\*\*kwds**

Keyword arguments to pass on to `pandas.DataFrame.plot()`.

**Returns** `axes` : `matplotlib.axes.Axes` or `numpy.ndarray`

Returns an ndarray when `subplots=True`.

**See also:**

`matplotlib.pyplot.plot` Plot y versus x as lines and/or markers.

## Examples

The following example shows the populations for some animals over the years.

```
>>> df = pd.DataFrame({
...     'pig': [20, 18, 489, 675, 1776],
...     'horse': [4, 25, 281, 600, 1900]
...     }, index=[1990, 1997, 2003, 2009, 2014])
>>> lines = df.plot.line()
```

An example with subplots, so an array of axes is returned.

```
>>> axes = df.plot.line(subplots=True)
>>> type(axes)
<class 'numpy.ndarray'>
```

The following example shows the relationship between both populations.

```
>>> lines = df.plot.line(x='pig', y='horse')
```

### 34.4.13.10 pandas.DataFrame.plot.pie

`DataFrame.plot.pie` (`y=None`, `**kwds`)

Generate a pie plot.

A pie plot is a proportional representation of the numerical data in a column. This function wraps `matplotlib.pyplot.pie()` for the specified column. If no column reference is passed and `subplots=True` a pie plot is drawn for each numerical column independently.

**Parameters** **y** : int or label, optional

Label or position of the column to plot. If not provided, `subplots=True` argument must be passed.

**\*\*kwds**

Keyword arguments to pass on to `pandas.DataFrame.plot()`.

**Returns** `axes` : `matplotlib.axes.Axes` or `np.ndarray` of them.

A NumPy array is returned when *subplots* is True.

See also:

**`Series.plot.pie`** Generate a pie plot for a Series.

**`DataFrame.plot`** Make plots of a DataFrame.

## Examples

In the example below we have a DataFrame with the information about planet's mass and radius. We pass the 'mass' column to the pie function to get a pie plot.

```
>>> df = pd.DataFrame({'mass': [0.330, 4.87, 5.97],
...                     'radius': [2439.7, 6051.8, 6378.1]},
...                     index=['Mercury', 'Venus', 'Earth'])
>>> plot = df.plot.pie(y='mass', figsize=(5, 5))
```

```
>>> plot = df.plot.pie(subplots=True, figsize=(6, 3))
```

### 34.4.13.11 pandas.DataFrame.plot.scatter

**DataFrame.plot.scatter** (*x*, *y*, *s=None*, *c=None*, *\*\*kws*)

Create a scatter plot with varying marker point size and color.

The coordinates of each point are defined by two dataframe columns and filled circles are used to represent each point. This kind of plot is useful to see complex correlations between two variables. Points could be for instance natural 2D coordinates like longitude and latitude in a map or, in general, any pair of metrics that can be plotted against each other.

**Parameters** **x** : int or str

The column name or column position to be used as horizontal coordinates for each point.

**y** : int or str

The column name or column position to be used as vertical coordinates for each point.

**s** : scalar or array\_like, optional

The size of each point. Possible values are:

- A single scalar so all points have the same size.
- A sequence of scalars, which will be used for each point's size recursively. For instance, when passing [2,14] all points size will be either 2 or 14, alternatively.

**c** : str, int or array\_like, optional

The color of each point. Possible values are:

- A single color string referred to by name, RGB or RGBA code, for instance 'red' or '#a98d19'.
- A sequence of color strings referred to by name, RGB or RGBA code, which will be used for each point's color recursively. For instance ['green', 'yellow'] all points will be filled in green or yellow, alternatively.

- A column name or position whose values will be used to color the marker points according to a colormap.

**\*\*kwargs**

Keyword arguments to pass on to `pandas.DataFrame.plot()`.

**Returns**

**axes** [`matplotlib.axes.Axes` or `numpy.ndarray` of them]

**See also:**

**matplotlib.pyplot.scatter** scatter plot using multiple input data formats.

## Examples

Let's see how to draw a scatter plot using coordinates from the values in a DataFrame's columns.

```
>>> df = pd.DataFrame([[5.1, 3.5, 0], [4.9, 3.0, 0], [7.0, 3.2, 1],
...                    [6.4, 3.2, 1], [5.9, 3.0, 2]],
...                    columns=['length', 'width', 'species'])
>>> ax1 = df.plot.scatter(x='length',
...                        y='width',
...                        c='DarkBlue')
```

And now with the color determined by a column as well.

```
>>> ax2 = df.plot.scatter(x='length',
...                        y='width',
...                        c='species',
...                        colormap='viridis')
```

<code>DataFrame.boxplot([column, by, ax, ...])</code>	Make a box plot from DataFrame columns.
<code>DataFrame.hist([column, by, grid, ...])</code>	Make a histogram of the DataFrame's.

## 34.4.14 Serialization / IO / Conversion

<code>DataFrame.from_csv(path[, header, sep, ...])</code>	(DEPRECATED) Read CSV file.
<code>DataFrame.from_dict(data[, orient, dtype, ...])</code>	Construct DataFrame from dict of array-like or dicts.
<code>DataFrame.from_items(items[, columns, orient])</code>	(DEPRECATED) Construct a dataframe from a list of tuples
<code>DataFrame.from_records(data[, index, ...])</code>	Convert structured or record ndarray to DataFrame
<code>DataFrame.info([verbose, buf, max_cols, ...])</code>	Print a concise summary of a DataFrame.
<code>DataFrame.to_parquet(fname[, engine, ...])</code>	Write a DataFrame to the binary parquet format.
<code>DataFrame.to_pickle(path[, compression, ...])</code>	Pickle (serialize) object to file.
<code>DataFrame.to_csv([path_or_buf, sep, na_rep, ...])</code>	Write DataFrame to a comma-separated values (csv) file
<code>DataFrame.to_hdf(path_or_buf, key, **kwargs)</code>	Write the contained data to an HDF5 file using HDFStore.
<code>DataFrame.to_sql(name, con[, schema, ...])</code>	Write records stored in a DataFrame to a SQL database.
<code>DataFrame.to_dict([orient, into])</code>	Convert the DataFrame to a dictionary.
<code>DataFrame.to_excel(excel_writer[, ...])</code>	Write DataFrame to an excel sheet

Continued on next page

Table 77 – continued from previous page

<code>DataFrame.to_json([path_or_buf, orient, ...])</code>	Convert the object to a JSON string.
<code>DataFrame.to_html([buf, columns, col_space, ...])</code>	Render a DataFrame as an HTML table.
<code>DataFrame.to_feather(fname)</code>	write out the binary feather-format for DataFrames
<code>DataFrame.to_latex([buf, columns, ...])</code>	Render an object to a tabular environment table.
<code>DataFrame.to_stata(fname[, convert_dates, ...])</code>	Export Stata binary dta files.
<code>DataFrame.to_msgpack([path_or_buf, encoding])</code>	msgpack (serialize) object to input file path
<code>DataFrame.to_gbq(destination_table, project_id)</code>	Write a DataFrame to a Google BigQuery table.
<code>DataFrame.to_records([index, convert_datetime64])</code>	Convert DataFrame to a NumPy record array.
<code>DataFrame.to_sparse([fill_value, kind])</code>	Convert to SparseDataFrame
<code>DataFrame.to_dense()</code>	Return dense representation of NDFrame (as opposed to sparse)
<code>DataFrame.to_string([buf, columns, ...])</code>	Render a DataFrame to a console-friendly tabular output.
<code>DataFrame.to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>DataFrame.style</code>	Property returning a Styler object containing methods for building a styled HTML representation fo the DataFrame.

### 34.4.15 Sparse

<code>SparseDataFrame.to_coo()</code>	Return the contents of the frame as a sparse SciPy COO matrix.
---------------------------------------	--

#### 34.4.15.1 pandas.SparseDataFrame.to\_coo

`SparseDataFrame.to_coo()`

Return the contents of the frame as a sparse SciPy COO matrix.

New in version 0.20.0.

**Returns** `coo_matrix` : `scipy.sparse.spmatrix`

If the caller is heterogeneous and contains booleans or objects, the result will be of `dtype=object`. See Notes.

#### Notes

The dtype will be the lowest-common-denominator type (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen.

e.g. If the dtypes are `float16` and `float32`, dtype will be upcast to `float32`. By `numpy.find_common_type` convention, mixing `int64` and `uint64` will result in a `float64` dtype.

## 34.5 Panel

### 34.5.1 Constructor

---

<code>Panel([data, items, major_axis, minor_axis, ...])</code>	(DEPRECATED) Represents wide format panel data, stored as 3-dimensional array
--	---

---

#### 34.5.1.1 pandas.Panel

**class** `pandas.Panel` (*data=None, items=None, major\_axis=None, minor\_axis=None, copy=False, dtype=None*)

Represents wide format panel data, stored as 3-dimensional array

Deprecated since version 0.20.0: The recommended way to represent 3-D data are with a MultiIndex on a DataFrame via the `to_frame()` method or with the `xarray` package. Pandas provides a `to_xarray()` method to automate this conversion.

**Parameters** **data** : ndarray (items x major x minor), or dict of DataFrames

**items** [Index or array-like] axis=0

**major\_axis** [Index or array-like] axis=1

**minor\_axis** [Index or array-like] axis=2

**dtype** [dtype, default None] Data type to force, otherwise infer

**copy** [boolean, default False] Copy data from inputs. Only affects DataFrame / 2d ndarray input

#### Attributes

<code>at</code>	Access a single value for a row/column label pair.
<code>axes</code>	Return index label(s) of the internal NDFrame
<code>blocks</code>	(DEPRECATED) Internal property, property synonym for <code>as_blocks()</code>
<code>dtypes</code>	Return the dtypes in the DataFrame.
<code>empty</code>	Indicator whether DataFrame is empty.
<code>ftypes</code>	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
<code>iat</code>	Access a single value for a row/column pair by integer position.
<code>iloc</code>	Purely integer-location based indexing for selection by position.
<code>items</code>	
<code>ix</code>	A primarily label-location based indexer, with integer position fallback.
<code>loc</code>	Access a group of rows and columns by label(s) or a boolean array.
<code>major_axis</code>	
<code>minor_axis</code>	

Continued on next page



Table 80 – continued from previous page

<i>ndim</i>	Return an int representing the number of axes / array dimensions.
<i>shape</i>	Return a tuple of axis dimensions
<i>size</i>	Return an int representing the number of elements in this object.
<i>values</i>	Return a Numpy representation of the DataFrame.

**pandas.Panel.at****Panel.at**

Access a single value for a row/column label pair.

Similar to `loc`, in that both provide label-based lookups. Use `at` if you only need to get or set a single value in a DataFrame or Series.

**Raises KeyError**

When label does not exist in DataFrame

**See also:**

**DataFrame.iat** Access a single value for a row/column pair by integer position

**DataFrame.loc** Access a group of rows and columns by label(s)

**Series.at** Access a single value using a label

**Examples**

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
   A  B  C
4  0  2  3
5  0  4  1
6 10 20 30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

## pandas.Panel.axes

### Panel.axes

Return index label(s) of the internal NDFrame

## pandas.Panel.blocks

### Panel.blocks

Internal property, property synonym for `as_blocks()`

Deprecated since version 0.21.0.

## pandas.Panel.dtypes

### Panel.dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See [the User Guide](#) for more.

#### Returns pandas.Series

The data type of each column.

See also:

[`pandas.DataFrame.ftypes`](#) dtype and sparsity information.

## Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float          float64
int            int64
datetime      datetime64[ns]
string         object
dtype: object
```

## pandas.Panel.empty

### Panel.empty

Indicator whether DataFrame is empty.

True if DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

#### Returns bool

If DataFrame is empty, return True, if not return False.

See also:

[`pandas.Series.dropna`](#), [`pandas.DataFrame.dropna`](#)

## Notes

If DataFrame contains only NaNs, it is still not considered empty. See the example below.

## Examples

An example of an actual empty DataFrame. Notice the index is empty:

```

>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True

```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```

>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
   A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True

```

## pandas.Panel.ftypes

### Panel.ftypes

Return the ftypes (indication of sparse/dense and dtype) in DataFrame.

This returns a Series with the data type of each column. The result's index is the original DataFrame's columns. Columns with mixed types are stored with the `object` dtype. See [the User Guide](#) for more.

#### Returns pandas.Series

The data type and indication of sparse/dense of each column.

#### See also:

**`pandas.DataFrame.dtypes`** Series with just dtype information.

**`pandas.SparseDataFrame`** Container for sparse tabular data.

## Notes

Sparse data should have the same dtypes as its dense representation.

## Examples

```
>>> import numpy as np
>>> arr = np.random.RandomState(0).randn(100, 4)
>>> arr[arr < .8] = np.nan
>>> pd.DataFrame(arr).ftypes
0    float64:dense
1    float64:dense
2    float64:dense
3    float64:dense
dtype: object
```

```
>>> pd.SparseDataFrame(arr).ftypes
0    float64:sparse
1    float64:sparse
2    float64:sparse
3    float64:sparse
dtype: object
```

## pandas.Panel.iat

### Panel.iat

Access a single value for a row/column pair by integer position.

Similar to `iloc`, in that both provide integer-based lookups. Use `iat` if you only need to get or set a single value in a DataFrame or Series.

#### Raises `IndexError`

When integer position is out of bounds

#### See also:

**`DataFrame.at`** Access a single value for a row/column label pair

**`DataFrame.loc`** Access a group of rows and columns by label(s)

**`DataFrame.iloc`** Access a group of rows and columns by integer position(s)

## Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                    columns=['A', 'B', 'C'])
>>> df
   A  B  C
0  0  2  3
1  0  4  1
2 10 20 30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

## pandas.Panel.iloc

### Panel.iloc

Purely integer-location based indexing for selection by position.

`.iloc[]` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

See more at [Selection by Position](#)

## pandas.Panel.items

### Panel.items

## pandas.Panel.ix

### Panel.ix

A primarily label-location based indexer, with integer position fallback.

Warning: Starting in 0.20.0, the `.ix` indexer is deprecated, in favor of the more strict `.iloc` and `.loc` indexers.

`.ix[]` supports mixed integer and label based access. It is primarily label based, but will fall back to integer positional access unless the corresponding axis is of integer type.

`.ix` is the most general indexer and will support any of the inputs in `.loc` and `.iloc`. `.ix` also supports floating point label schemes. `.ix` is exceptionally useful when dealing with mixed positional and label based hierarchical indexes.

However, when an axis is integer based, ONLY label based access and not positional access is supported. Thus, in such cases, it's usually better to be explicit and use `.iloc` or `.loc`.

See more at [Advanced Indexing](#).

## pandas.Panel.loc

### Panel.loc

Access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a *label* of the index, and **never** as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- A slice object with labels, e.g. 'a':'f'.

**Warning:** Note that contrary to usual python slices, **both** the start and the stop are included

- A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
- A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above)

See more at [Selection by Label](#)

#### Raises KeyError:

when any items are not found

See also:

**DataFrame.at** Access a single value for a row/column label pair

**DataFrame.iloc** Access group of rows and columns by integer position(s)

**DataFrame.xs** Returns a cross-section (row(s) or column(s)) from the Series/DataFrame.

**Series.loc** Access group of values using labels

## Examples

### Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=['cobra', 'viper', 'sidewinder'],
...                    columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
cobra	1	2
viper	4	5
sidewinder	7	8

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using `[[]]` returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
          max_speed  shield
viper              4      5
sidewinder         7      8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra      1
viper      4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
          max_speed  shield
sidewinder         7      8
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
          max_speed  shield
sidewinder         7      8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
          max_speed
sidewinder         7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
          max_speed  shield
sidewinder         7      8
```

## Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
          max_speed  shield
cobra              1      2
viper              4     50
sidewinder         7     50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
```

	max_speed	shield
cobra	10	10
viper	4	50
sidewinder	7	50

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
```

	max_speed	shield
cobra	30	10
viper	30	50
sidewinder	30	50

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
```

	max_speed	shield
cobra	30	10
viper	0	0
sidewinder	0	0

### Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...                    index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
```

	max_speed	shield
7	1	2
8	4	5
9	7	8

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
```

	max_speed	shield
7	1	2
8	4	5
9	7	8

### Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
```

(continues on next page)



(continued from previous page)

```
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
```

		max_speed	shield
cobra	mark i	12	2
	mark ii	0	4
sidewinder	mark i	10	20
	mark ii	1	4
viper	mark ii	7	1
	mark iii	16	36

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
```

	max_speed	shield
mark i	12	2
mark ii	0	4

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield       2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using [ [] ] returns a DataFrame.

```
>>> df.loc[['cobra', 'mark ii']]
```

		max_speed	shield
cobra	mark ii	0	4

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
```

		max_speed	shield
cobra	mark i	12	2
	mark ii	0	4
sidewinder	mark i	10	20
	mark ii	1	4
viper	mark ii	7	1
	mark iii	16	36

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
      max_speed  shield
cobra      mark i      12      2
           mark ii      0      4
sidewinder mark i      10     20
           mark ii      1      4
viper      mark ii      7      1
```

### pandas.Panel.major\_axis

Panel.**major\_axis**

### pandas.Panel.minor\_axis

Panel.**minor\_axis**

### pandas.Panel.ndim

Panel.**ndim**

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

**See also:**

ndarray.ndim

### Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

### pandas.Panel.shape

Panel.**shape**

Return a tuple of axis dimensions

### pandas.Panel.size

Panel.**size**

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

**See also:**`ndarray.size`**Examples**

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

**pandas.Panel.values****Panel.values**

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

**Returns** `numpy.ndarray`

The values of the DataFrame.

**See also:**

[`pandas.DataFrame.index`](#) Retrieve the index labels

[`pandas.DataFrame.columns`](#) Retrieving the column names

**Notes**

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By `numpy.find_common_type()` convention, mixing int64 and uint64 will result in a float64 dtype.

**Examples**

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age': [ 3, 29],
...                     'height': [94, 170],
...                     'weight': [31, 115]})
>>> df
   age  height  weight
0    3     94     31
1   29    170    115
>>> df.dtypes
```

(continues on next page)

(continued from previous page)

```

age      int64
height   int64
weight   int64
dtype: object
>>> df.values
array([[ 3,  94,  31],
       [29, 170, 115]], dtype=int64)

```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```

>>> df2 = pd.DataFrame([('parrot', 24.0, 'second'),
...                      ('lion', 80.5, 1),
...                      ('monkey', np.nan, None)],
...                     columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name      object
max_speed float64
rank      object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)

```

is_copy	
---------	--

## Methods

<code>abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>add(other[, axis])</code>	Addition of series and other, element-wise (binary operator <i>add</i> ).
<code>add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>align(other, **kwargs)</code>	Align two objects on their axes with the specified join method for each axis Index
<code>all([axis, bool_only, skipna, level])</code>	Return whether all elements are True, potentially over an axis.
<code>any([axis, bool_only, skipna, level])</code>	Return whether any element is True over requested axis.
<code>apply(func[, axis])</code>	Applies function along axis (or axes) of the Panel
<code>as_blocks([copy])</code>	(DEPRECATED) Convert the frame to a dict of dtype -> Constructor Types that each has a homogeneous dtype.
<code>as_matrix()</code>	Convert the frame to its Numpy-array representation.
<code>asfreq(freq[, method, how, normalize, ...])</code>	Convert TimeSeries to specified frequency.
<code>asof(where[, subset])</code>	The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

Continued on next page

Table 81 – continued from previous page

<code>astype(dtype[, copy, errors])</code>	Cast a pandas object to a specified dtype <code>dtype</code> .
<code>at_time(time[, asof])</code>	Select values at particular time of day (e.g.
<code>between_time(start_time, end_time[, ...])</code>	Select values between particular times of the day (e.g., 9:00-9:30 AM).
<code>bfill([axis, inplace, limit, downcast])</code>	Synonym for <code>DataFrame.fillna(method='bfill')</code>
<code>bool()</code>	Return the bool of a single element PandasObject.
<code>clip([lower, upper, axis, inplace])</code>	Trim values at input threshold(s).
<code>clip_lower(threshold[, axis, inplace])</code>	Return copy of the input with values below a threshold truncated.
<code>clip_upper(threshold[, axis, inplace])</code>	Return copy of input with values above given value(s) truncated.
<code>compound([axis, skipna, level])</code>	Return the compound percentage of the values for the requested axis
<code>conform(frame[, axis])</code>	Conform input DataFrame to align with chosen axis pair.
<code>consolidate([inplace])</code>	(DEPRECATED) Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray).
<code>convert_objects([convert_dates, ...])</code>	(DEPRECATED) Attempt to infer better dtype for object columns.
<code>copy([deep])</code>	Make a copy of this object’s indices and data.
<code>count([axis])</code>	Return number of observations over requested axis.
<code>cummax([axis, skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>cummin([axis, skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>cumprod([axis, skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>cumsum([axis, skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>describe([percentiles, include, exclude])</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.
<code>div(other[, axis])</code>	Floating division of series and other, element-wise (binary operator <code>truediv</code> ).
<code>divide(other[, axis])</code>	Floating division of series and other, element-wise (binary operator <code>truediv</code> ).
<code>dropna([axis, how, inplace])</code>	Drop 2D from panel, holding passed axis constant
<code>eq(other[, axis])</code>	Wrapper for comparison method <code>eq</code>
<code>equals(other)</code>	Determines if two NDFrame objects contain the same elements.
<code>ffill([axis, inplace, limit, downcast])</code>	Synonym for <code>DataFrame.fillna(method='ffill')</code>
<code>fillna([value, method, axis, inplace, ...])</code>	Fill NA/NaN values using the specified method
<code>filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>first_valid_index()</code>	Return index for first non-NA/null value.

Continued on next page

Table 81 – continued from previous page

<code>floordiv</code> ( <i>other</i> [, <i>axis</i> ])	Integer division of series and other, element-wise (binary operator <i>floordiv</i> ).
<code>fromDict</code> ( <i>data</i> [, <i>intersect</i> , <i>orient</i> , <i>dtype</i> ])	Construct Panel from dict of DataFrame objects
<code>from_dict</code> ( <i>data</i> [, <i>intersect</i> , <i>orient</i> , <i>dtype</i> ])	Construct Panel from dict of DataFrame objects
<code>ge</code> ( <i>other</i> [, <i>axis</i> ])	Wrapper for comparison method <i>ge</i>
<code>get</code> ( <i>key</i> [, <i>default</i> ])	Get item from object for given key (DataFrame column, Panel slice, etc.).
<code>get_dtype_counts</code> ()	Return counts of unique dtypes in this object.
<code>get_ftype_counts</code> ()	(DEPRECATED) Return counts of unique ftypes in this object.
<code>get_value</code> (*args, **kwargs)	(DEPRECATED) Quickly retrieve single value at (item, major, minor) location
<code>get_values</code> ()	Return an ndarray after converting sparse values to dense.
<code>groupby</code> ( <i>function</i> [, <i>axis</i> ])	Group data on given axis, returning GroupBy object
<code>gt</code> ( <i>other</i> [, <i>axis</i> ])	Wrapper for comparison method <i>gt</i>
<code>head</code> ([ <i>n</i> ])	Return the first <i>n</i> rows.
<code>infer_objects</code> ()	Attempt to infer better dtypes for object columns.
<code>interpolate</code> ([ <i>method</i> , <i>axis</i> , <i>limit</i> , <i>inplace</i> , ...])	Interpolate values according to different methods.
<code>isna</code> ()	Detect missing values.
<code>isnull</code> ()	Detect missing values.
<code>iteritems</code> ()	Iterate over (label, values) on info axis
<code>join</code> ( <i>other</i> [, <i>how</i> , <i>lsuffix</i> , <i>rsuffix</i> ])	Join items with other Panel either on major and minor axes column
<code>keys</code> ()	Get the ‘info axis’ (see Indexing for more)
<code>kurt</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> , <i>numeric_only</i> ])	Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).
<code>kurtosis</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> , <i>numeric_only</i> ])	Return unbiased kurtosis over requested axis using Fisher’s definition of kurtosis (kurtosis of normal == 0.0).
<code>last</code> ( <i>offset</i> )	Convenience method for subsetting final periods of time series data based on a date offset.
<code>last_valid_index</code> ()	Return index for last non-NA/null value.
<code>le</code> ( <i>other</i> [, <i>axis</i> ])	Wrapper for comparison method <i>le</i>
<code>lt</code> ( <i>other</i> [, <i>axis</i> ])	Wrapper for comparison method <i>lt</i>
<code>mad</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> ])	Return the mean absolute deviation of the values for the requested axis
<code>major_xs</code> ( <i>key</i> )	Return slice of panel along major axis
<code>mask</code> ( <i>cond</i> [, <i>other</i> , <i>inplace</i> , <i>axis</i> , <i>level</i> , ...])	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is False and otherwise are from <i>other</i> .
<code>max</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> , <i>numeric_only</i> ])	This method returns the maximum of the values in the object.
<code>mean</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> , <i>numeric_only</i> ])	Return the mean of the values for the requested axis
<code>median</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> , <i>numeric_only</i> ])	Return the median of the values for the requested axis
<code>min</code> ([ <i>axis</i> , <i>skipna</i> , <i>level</i> , <i>numeric_only</i> ])	This method returns the minimum of the values in the object.
<code>minor_xs</code> ( <i>key</i> )	Return slice of panel along minor axis

Continued on next page

Table 81 – continued from previous page

<code>mod(other[, axis])</code>	Modulo of series and other, element-wise (binary operator <i>mod</i> ).
<code>mul(other[, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>multiply(other[, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<code>ne(other[, axis])</code>	Wrapper for comparison method <i>ne</i>
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>pct_change([periods, fill_method, limit, freq])</code>	Percentage change between the current and a prior element.
<code>pipe(func, *args, **kwargs)</code>	Apply <i>func</i> (self, *args, **kwargs)
<code>pop(item)</code>	Return item and drop from frame.
<code>pow(other[, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>pow</i> ).
<code>prod([axis, skipna, level, numeric_only, ...])</code>	Return the product of the values for the requested axis
<code>product([axis, skipna, level, numeric_only, ...])</code>	Return the product of the values for the requested axis
<code>radd(other[, axis])</code>	Addition of series and other, element-wise (binary operator <i>radd</i> ).
<code>rank([axis, method, numeric_only, ...])</code>	Compute numerical data ranks (1 through n) along axis.
<code>rdiv(other[, axis])</code>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<code>reindex(*args, **kwargs)</code>	Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_axis(labels[, axis, method, level, ...])</code>	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>reindex_like(other[, method, copy, limit, ...])</code>	Return an object with matching indices to myself.
<code>rename([items, major_axis, minor_axis])</code>	Alter axes input function or functions.
<code>rename_axis(mapper[, axis, copy, inplace])</code>	Alter the name of the index or columns.
<code>replace([to_replace, value, inplace, limit, ...])</code>	Replace values given in <i>to_replace</i> with <i>value</i> .
<code>resample(rule[, how, axis, fill_method, ...])</code>	Convenience method for frequency conversion and resampling of time series.
<code>rfloordiv(other[, axis])</code>	Integer division of series and other, element-wise (binary operator <i>rfloordiv</i> ).
<code>rmod(other[, axis])</code>	Modulo of series and other, element-wise (binary operator <i>rmod</i> ).
<code>rmul(other[, axis])</code>	Multiplication of series and other, element-wise (binary operator <i>rmul</i> ).
<code>round([decimals])</code>	Round each value in Panel to a specified number of decimal places.
<code>rpow(other[, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>rpow</i> ).
<code>rsub(other[, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>rsub</i> ).
<code>rtruediv(other[, axis])</code>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).

Continued on next page

Table 81 – continued from previous page

<code>sample([n, frac, replace, weights, ...])</code>	Return a random sample of items from an axis of object.
<code>select(crit[, axis])</code>	(DEPRECATED) Return data corresponding to axis labels matching criteria
<code>sem([axis, skipna, level, ddof, numeric_only])</code>	Return unbiased standard error of the mean over requested axis.
<code>set_axis(labels[, axis, inplace])</code>	Assign desired index to given axis.
<code>set_value(*args, **kwargs)</code>	(DEPRECATED) Quickly set single value at (item, major, minor) location
<code>shift([periods, freq, axis])</code>	Shift index by desired number of periods with an optional time freq.
<code>skew([axis, skipna, level, numeric_only])</code>	Return unbiased skew over requested axis Normalized by N-1
<code>slice_shift([periods, axis])</code>	Equivalent to <i>shift</i> without copying data.
<code>sort_index([axis, level, ascending, ...])</code>	Sort object by labels (along an axis)
<code>sort_values([by, axis, ascending, inplace, ...])</code>	NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.
<code>squeeze([axis])</code>	Squeeze length 1 dimensions.
<code>std([axis, skipna, level, ddof, numeric_only])</code>	Return sample standard deviation over requested axis.
<code>sub(other[, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>subtract(other[, axis])</code>	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<code>sum([axis, skipna, level, numeric_only, ...])</code>	Return the sum of the values for the requested axis
<code>swapaxes(axis1, axis2[, copy])</code>	Interchange axes and swap values axes appropriately
<code>swaplevel([i, j, axis])</code>	Swap levels i and j in a MultiIndex on a particular axis
<code>tail([n])</code>	Return the last <i>n</i> rows.
<code>take(indices[, axis, convert, is_copy])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>to_clipboard([excel, sep])</code>	Copy object to the system clipboard.
<code>to_dense()</code>	Return dense representation of NDFrame (as opposed to sparse)
<code>to_excel(path[, na_rep, engine])</code>	Write each DataFrame in Panel to a separate excel sheet
<code>to_frame([filter_observations])</code>	Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.
<code>to_hdf(path_or_buf, key, **kwargs)</code>	Write the contained data to an HDF5 file using HDF-Store.
<code>to_json([path_or_buf, orient, date_format, ...])</code>	Convert the object to a JSON string.
<code>to_latex([buf, columns, col_space, header, ...])</code>	Render an object to a tabular environment table.
<code>to_msgpack([path_or_buf, encoding])</code>	msgpack (serialize) object to input file path
<code>to_pickle(path[, compression, protocol])</code>	Pickle (serialize) object to file.
<code>to_sparse(*args, **kwargs)</code>	NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.

Continued on next page



Table 81 – continued from previous page

<code>to_sql(name, con[, schema, if_exists, ...])</code>	Write records stored in a DataFrame to a SQL database.
<code>to_xarray()</code>	Return an xarray object from the pandas object.
<code>transpose(*args, **kwargs)</code>	Permute the dimensions of the Panel
<code>truediv(other[, axis])</code>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<code>truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.
<code>tshift([periods, freq, axis])</code>	Shift the time index, using the index's frequency if available.
<code>tz_convert(tz[, axis, level, copy])</code>	Convert tz-aware axis to target time zone.
<code>tz_localize(tz[, axis, level, copy, ambiguous])</code>	Localize tz-naive TimeSeries to target time zone.
<code>update(other[, join, overwrite, ...])</code>	Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.
<code>var([axis, skipna, level, ddof, numeric_only])</code>	Return unbiased variance over requested axis.
<code>where(cond[, other, inplace, axis, level, ...])</code>	Return an object of same shape as self and whose corresponding entries are from self where <i>cond</i> is True and otherwise are from <i>other</i> .
<code>xs(key[, axis])</code>	Return slice of panel along selected axis

**pandas.Panel.abs**`Panel.abs()`

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

**Returns abs**

Series/DataFrame containing the absolute value of each element.

**See also:**`numpy.absolute` calculate the absolute value element-wise.**Notes**For complex inputs,  $1.2 + 1j$ , the absolute value is  $\sqrt{a^2 + b^2}$ .**Examples**

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0    1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using `argsort` (from [StackOverflow](#)).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
   a  b  c
0  4 10 100
1  5 20  50
2  6 30 -30
3  7 40 -50
>>> df.loc[(df.c - 43).abs().argsort()]
   a  b  c
1  5 20  50
0  4 10 100
2  6 30 -30
3  7 40 -50
```

## **pandas.Panel.add**

`Panel.add(other, axis=0)`

Addition of series and other, element-wise (binary operator *add*). Equivalent to `panel + other`.

### **Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### **Returns**

**Panel**

**See also:**

[\*Panel.radd\*](#)

## **pandas.Panel.add\_prefix**

`Panel.add_prefix(prefix)`

Prefix labels with string *prefix*.

For Series, the row labels are prefixed. For DataFrame, the column labels are prefixed.

**Parameters** `prefix` : str

The string to add before each label.

**Returns** Series or DataFrame

New Series or DataFrame with updated labels.

**See also:**

[`Series.add\_suffix`](#) Suffix row labels with string *suffix*.

[`DataFrame.add\_suffix`](#) Suffix column labels with string *suffix*.

## Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_prefix('item_')
item_0    1
item_1    2
item_2    3
item_3    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_prefix('col_')
   col_A  col_B
0      1      3
1      2      4
2      3      5
3      4      6
```

## pandas.Panel.add\_suffix

`Panel.add_suffix` (*suffix*)

Suffix labels with string *suffix*.

For Series, the row labels are suffixed. For DataFrame, the column labels are suffixed.

**Parameters** `suffix` : str

The string to add after each label.

### Returns Series or DataFrame

New Series or DataFrame with updated labels.

### See also:

[`Series.add\_prefix`](#) Prefix row labels with string *prefix*.

[`DataFrame.add\_prefix`](#) Prefix column labels with string *prefix*.

### Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.add_suffix('_item')
0_item    1
1_item    2
2_item    3
3_item    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4], 'B': [3, 4, 5, 6]})
>>> df
   A  B
0  1  3
1  2  4
2  3  5
3  4  6
```

```
>>> df.add_suffix('_col')
   A_col  B_col
0      1     3
1      2     4
2      3     5
3      4     6
```

## pandas.Panel.align

`Panel.align` (*other*, *\*\*kwargs*)

Align two objects on their axes with the specified join method for each axis Index

### Parameters

**other** [DataFrame or Series]

**join** [{‘outer’, ‘inner’, ‘left’, ‘right’}, default ‘outer’]

**axis** : allowed axis of the other object, default None

Align on index (0), columns (1), or both (None)

**level** : int or level name, default None

Broadcast across a level, matching Index values on the passed MultiIndex level

**copy** : boolean, default True

Always returns new objects. If copy=False and no reindexing is required then original objects are returned.

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any “compatible” value

**method** [str, default None]

**limit** [int, default None]

**fill\_axis** : int or labels for object, default 0

Filling axis, method and limit

**broadcast\_axis** : int or labels for object, default None

Broadcast values along this axis, if aligning two objects of different dimensions

**Returns** (**left**, **right**) : (NDFrame, type of other)

Aligned objects

## pandas.Panel.all

`Panel.all` (*axis=0*, *bool\_only=None*, *skipna=True*, *level=None*, *\*\*kwargs*)

Return whether all elements are True, potentially over an axis.

Returns True if all elements within a series or along a Dataframe axis are non-zero, not-empty or not-False.

**Parameters** **axis** : {0 or ‘index’, 1 or ‘columns’, None}, default 0

Indicate which axis or axes should be reduced.

- 0 / ‘index’ : reduce the index, return a Series whose index is the original column labels.
- 1 / ‘columns’ : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**\*\*kwargs** : any, default None

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**all** [DataFrame or Panel (if level specified)]

See also:

[`pandas.Series.all`](#) Return True if all elements are True

[`pandas.DataFrame.any`](#) Return True if one (or more) elements are True

## Examples

### Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
```

### DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1  col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1    True
col2   False
dtype: bool
```

Specify `axis='columns'` to check if row-wise values all return True.

```
>>> df.all(axis='columns')
0     True
1    False
dtype: bool
```

Or `axis=None` for whether every value is True.

```
>>> df.all(axis=None)
False
```

### `pandas.Panel.any`

`Panel.any` (*axis=0*, *bool\_only=None*, *skipna=True*, *level=None*, *\*\*kwargs*)

Return whether any element is True over requested axis.

Unlike `DataFrame.all()`, this performs an *or* operation. If any of the values along the specified axis is True, this will return True.

**Parameters** `axis` : {0 or 'index', 1 or 'columns', None}, default 0

Indicate which axis or axes should be reduced.

- 0 / 'index' : reduce the index, return a Series whose index is the original column labels.
- 1 / 'columns' : reduce the columns, return a Series whose index is the original index.
- None : reduce all axes, return a scalar.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame.

**bool\_only** : boolean, default None

Include only boolean columns. If None, will attempt to use everything, then use only boolean data. Not implemented for Series.

**\*\*kwargs** : any, default None

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**any** [DataFrame or Panel (if level specified)]

**See also:**

`pandas.DataFrame.all` Return whether all elements are True.

## Examples

### Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([True, False]).any()
True
```

### DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A      True
B      True
C     False
dtype: bool
```

Aggregating over the columns.

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 2]})
>>> df
   A  B
0  True  1
1 False  2
```

```
>>> df.any(axis='columns')
0      True
1      True
dtype: bool
```

```
>>> df = pd.DataFrame({"A": [True, False], "B": [1, 0]})
>>> df
   A  B
0  True  1
1 False  0
```

```
>>> df.any(axis='columns')
0      True
1     False
dtype: bool
```

Aggregating over the entire DataFrame with `axis=None`.

```
>>> df.any(axis=None)
True
```

`any` for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

## pandas.Panel.apply

`Panel.apply(func, axis='major', **kwargs)`

Applies function along axis (or axes) of the Panel

**Parameters** `func` : function

Function to apply to each combination of ‘other’ axes e.g. if `axis = ‘items’`, the combination of `major_axis/minor_axis` will each be passed as a Series; if `axis = (‘items’, ‘major’)`, DataFrames of items & major axis will be passed

**axis** : {‘items’, ‘minor’, ‘major’}, or {0, 1, 2}, or a tuple with two axes

**Additional keyword arguments will be passed as keywords to the function**



**Returns****result** [Panel, DataFrame, or Series]**Examples**

Returns a Panel with the square root of each element

```
>>> p = pd.Panel(np.random.rand(4, 3, 2))
>>> p.apply(np.sqrt)
```

Equivalent to `p.sum(1)`, returning a DataFrame

```
>>> p.apply(lambda x: x.sum(), axis=1)
```

Equivalent to previous:

```
>>> p.apply(lambda x: x.sum(), axis='major')
```

Return the shapes of each DataFrame over axis 2 (i.e the shapes of items x major), as a Series

```
>>> p.apply(lambda x: x.shape, axis=(0, 1))
```

**pandas.Panel.as\_blocks**`Panel.as_blocks` (*copy=True*)

Convert the frame to a dict of dtype -&gt; Constructor Types that each has a homogeneous dtype.

Deprecated since version 0.21.0.

**NOTE: the dtypes of the blocks WILL BE PRESERVED HERE (unlike in `as_matrix`)****Parameters****copy** [boolean, default True]**Returns****values** [a dict of dtype -> Constructor Types]**pandas.Panel.as\_matrix**`Panel.as_matrix()`

Convert the frame to its Numpy-array representation.

Deprecated since version 0.23.0: Use `DataFrame.values()` instead.**Parameters columns: list, optional, default:None**

If None, return all columns, otherwise, returns specified columns.

**Returns values : ndarray**

If the caller is heterogeneous and contains booleans or objects, the result will be of dtype=object. See Notes.

**See also:**`pandas.DataFrame.values`

## Notes

Return is NOT a Numpy-matrix, rather, a Numpy-array.

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcase to int32. By `numpy.find_common_type` convention, mixing int64 and uint64 will result in a float64 dtype.

This method is provided for backwards compatibility. Generally, it is recommended to use `‘.values’`.

## pandas.Panel.asfreq

`Panel.asfreq(freq, method=None, how=None, normalize=False, fill_value=None)`

Convert TimeSeries to specified frequency.

Optionally provide filling method to pad/backfill missing values.

Returns the original data conformed to a new index with the specified frequency. `resample` is more appropriate if an operation, such as summarization, is necessary to represent the data at the new frequency.

### Parameters

**freq** [DateOffset object, or string]

**method** : {‘backfill’/‘bfill’, ‘pad’/‘ffill’}, default None

Method to use for filling holes in reindexed Series (note this does not fill NaNs that already were present):

- ‘pad’ / ‘ffill’: propagate last valid observation forward to next valid
- ‘backfill’ / ‘bfill’: use NEXT valid observation to fill

**how** : {‘start’, ‘end’}, default end

For PeriodIndex only, see PeriodIndex.asfreq

**normalize** : bool, default False

Whether to reset output index to midnight

**fill\_value**: scalar, optional

Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

### Returns

**converted** [type of caller]

**See also:**

[`reindex`](#)

## Notes

To learn more about the frequency strings, please see [this link](#).

## Examples

Start by creating a series with 4 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=4, freq='T')
>>> series = pd.Series([0.0, None, 2.0, 3.0], index=index)
>>> df = pd.DataFrame({'s':series})
>>> df
```

	s
2000-01-01 00:00:00	0.0
2000-01-01 00:01:00	NaN
2000-01-01 00:02:00	2.0
2000-01-01 00:03:00	3.0

Upsample the series into 30 second bins.

```
>>> df.upsample(freq='30S')
```

	s
2000-01-01 00:00:00	0.0
2000-01-01 00:00:30	NaN
2000-01-01 00:01:00	NaN
2000-01-01 00:01:30	NaN
2000-01-01 00:02:00	2.0
2000-01-01 00:02:30	NaN
2000-01-01 00:03:00	3.0

Upsample again, providing a fill value.

```
>>> df.upsample(freq='30S', fill_value=9.0)
```

	s
2000-01-01 00:00:00	0.0
2000-01-01 00:00:30	9.0
2000-01-01 00:01:00	NaN
2000-01-01 00:01:30	9.0
2000-01-01 00:02:00	2.0
2000-01-01 00:02:30	9.0
2000-01-01 00:03:00	3.0

Upsample again, providing a method.

```
>>> df.upsample(freq='30S', method='bfill')
```

	s
2000-01-01 00:00:00	0.0
2000-01-01 00:00:30	NaN
2000-01-01 00:01:00	NaN
2000-01-01 00:01:30	2.0
2000-01-01 00:02:00	2.0
2000-01-01 00:02:30	3.0
2000-01-01 00:03:00	3.0

## pandas.Panel.asof

`Panel.asof` (where, *subset=None*)

The last row without any NaN is taken (or the last row without NaN considering only the subset of columns in the case of a DataFrame)

New in version 0.19.0: For DataFrame

If there is no good value, NaN is returned for a Series a Series of NaN values for a DataFrame

#### Parameters

**where** [date or array of dates]

**subset** : string or list of strings, default None

if not None use these columns for NaN propagation

#### Returns where is scalar

- value or NaN if input is Series
- Series if input is DataFrame

**where is Index: same shape object as input**

See also:

*merge\_asof*

#### Notes

Dates are assumed to be sorted Raises if this is not the case

### pandas.Panel.astype

`Panel.astype(dtype, copy=True, errors='raise', **kwargs)`  
Cast a pandas object to a specified dtype dtype.

**Parameters dtype** : data type, or dict of column name -> data type

Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, ...}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

**copy** : bool, default True.

Return a copy when copy=True (be very careful setting copy=False as changes to values then may propagate to other pandas objects).

**errors** : {'raise', 'ignore'}, default 'raise'.

Control raising of exceptions on invalid data for provided dtype.

- `raise` : allow exceptions to be raised
- `ignore` : suppress exceptions. On error return original object

New in version 0.20.0.

**raise\_on\_error** : raise on invalid input

Deprecated since version 0.20.0: Use `errors` instead

**kwargs** [keyword arguments to pass on to the constructor]

#### Returns

**casted** [type of caller]

See also:

`pandas.to_datetime` Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Convert argument to a numeric type.

`numpy.ndarray.astype` Cast a numpy array to a specified type.

## Examples

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> ser.astype('category', ordered=True, categories=[2, 1])
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Note that using `copy=False` and changing data on a new pandas object may propagate changes:

```
>>> s1 = pd.Series([1,2])
>>> s2 = s1.astype('int64', copy=False)
>>> s2[0] = 10
>>> s1 # note that s1[0] has changed too
0    10
1     2
dtype: int64
```

## pandas.Panel.at\_time

`Panel.at_time` (*time*, *asof=False*)

Select values at particular time of day (e.g. 9:30AM).

### Parameters

**time** [datetime.time or string]

**Returns**

**values\_at\_time** [type of caller]

**Raises** `TypeError`

If the index is not a `DatetimeIndex`

**See also:**

**`between_time`** Select values between particular times of the day

**`first`** Select initial periods of time series based on a date offset

**`last`** Select final periods of time series based on a date offset

**`DatetimeIndex.indexer_at_time`** Get just the index locations for values at particular time of the day

**Examples**

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='12H')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09 00:00:00	1
2018-04-09 12:00:00	2
2018-04-10 00:00:00	3
2018-04-10 12:00:00	4

```
>>> ts.at_time('12:00')
```

	A
2018-04-09 12:00:00	2
2018-04-10 12:00:00	4

**pandas.Panel.between\_time**

`Panel.between_time` (*start\_time*, *end\_time*, *include\_start=True*, *include\_end=True*)

Select values between particular times of the day (e.g., 9:00-9:30 AM).

By setting *start\_time* to be later than *end\_time*, you can get the times that are *not* between the two times.

**Parameters**

**start\_time** [datetime.time or string]

**end\_time** [datetime.time or string]

**include\_start** [boolean, default True]

**include\_end** [boolean, default True]

**Returns**

**values\_between\_time** [type of caller]

**Raises** `TypeError`

If the index is not a `DatetimeIndex`

See also:

**`at_time`** Select values at a particular time of the day

**`first`** Select initial periods of time series based on a date offset

**`last`** Select final periods of time series based on a date offset

**`DatetimeIndex.indexer_between_time`** Get just the index locations for values between particular times of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='1D20min')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
```

	A
2018-04-09 00:00:00	1
2018-04-10 00:20:00	2
2018-04-11 00:40:00	3
2018-04-12 01:00:00	4

```
>>> ts.between_time('0:15', '0:45')
A
2018-04-10 00:20:00  2
2018-04-11 00:40:00  3
```

You get the times that are *not* between two times by setting `start_time` later than `end_time`:

```
>>> ts.between_time('0:45', '0:15')
A
2018-04-09 00:00:00  1
2018-04-12 01:00:00  4
```

## pandas.Panel.bfill

**`Panel.bfill`** (*axis=None, inplace=False, limit=None, downcast=None*)

Synonym for `DataFrame.fillna(method='bfill')`

## pandas.Panel.bool

**`Panel.bool()`**

Return the bool of a single element PandasObject.

This must be a boolean scalar value, either True or False. Raise a `ValueError` if the `PandasObject` does not have exactly 1 element, or that element is not boolean

## pandas.Panel.clip

**`Panel.clip`** (*lower=None, upper=None, axis=None, inplace=False, \*args, \*\*kwargs*)

Trim values at input threshold(s).

Assigns values outside boundary to boundary values. Thresholds can be singular values or array like, and in the latter case the clipping is performed element-wise in the specified axis.

**Parameters** **lower** : float or array\_like, default None

Minimum threshold value. All values below this threshold will be set to it.

**upper** : float or array\_like, default None

Maximum threshold value. All values above this threshold will be set to it.

**axis** : int or string axis name, optional

Align object with lower and upper along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data.

New in version 0.21.0.

**\*args, \*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns** **Series or DataFrame**

Same type as calling object with the values outside the clip boundaries replaced

**See also:**

***clip\_lower*** Clip values below specified threshold(s).

***clip\_upper*** Clip values above specified threshold(s).

## Examples

```
>>> data = {'col_0': [9, -3, 0, -1, 5], 'col_1': [-2, -7, 6, 8, -5]}
>>> df = pd.DataFrame(data)
>>> df
   col_0  col_1
0      9    -2
1     -3    -7
2      0     6
3     -1     8
4      5    -5
```

Clips per column using lower and upper thresholds:

```
>>> df.clip(-4, 6)
   col_0  col_1
0      6    -2
1     -3    -4
2      0     6
3     -1     6
4      5    -4
```

Clips using specific lower and upper thresholds per column element:



```
>>> t = pd.Series([2, -4, -1, 6, 3])
>>> t
0    2
1   -4
2   -1
3    6
4    3
dtype: int64
```

```
>>> df.clip(t, t + 4, axis=0)
   col_0  col_1
0      6      2
1     -3     -4
2      0      3
3      6      8
4      5      3
```

### pandas.Panel.clip\_lower

`Panel.clip_lower` (*threshold*, *axis=None*, *inplace=False*)

Return copy of the input with values below a threshold truncated.

**Parameters** *threshold* : numeric or array-like

Minimum value allowed. All values below threshold will be set to this value.

- float : every value is compared to *threshold*.
- array-like : The shape of *threshold* should match the object it's compared to. When *self* is a Series, *threshold* should be the length. When *self* is a DataFrame, *threshold* should 2-D and the same shape as *self* for *axis=None*, or 1-D and the same length as the axis being compared.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Align *self* with *threshold* along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data.

New in version 0.21.0.

#### Returns

**clipped** [same type as input]

**See also:**

**Series.clip** Return copy of input with values below and above thresholds truncated.

**Series.clip\_upper** Return copy of input with values above threshold truncated.

### Examples

Series single threshold clipping:

```
>>> s = pd.Series([5, 6, 7, 8, 9])
>>> s.clip_lower(8)
0      8
1      8
2      8
3      8
4      9
dtype: int64
```

Series clipping element-wise using an array of thresholds. *threshold* should be the same length as the Series.

```
>>> elemwise_thresholds = [4, 8, 7, 2, 5]
>>> s.clip_lower(elemwise_thresholds)
0      5
1      8
2      7
3      8
4      9
dtype: int64
```

DataFrames can be compared to a scalar.

```
>>> df = pd.DataFrame({"A": [1, 3, 5], "B": [2, 4, 6]})
>>> df
   A  B
0  1  2
1  3  4
2  5  6
```

```
>>> df.clip_lower(3)
   A  B
0  3  3
1  3  4
2  5  6
```

Or to an array of values. By default, *threshold* should be the same shape as the DataFrame.

```
>>> df.clip_lower(np.array([[3, 4], [2, 2], [6, 2]]))
   A  B
0  3  4
1  3  4
2  6  6
```

Control how *threshold* is broadcast with *axis*. In this case *threshold* should be the same length as the axis specified by *axis*.

```
>>> df.clip_lower(np.array([3, 3, 5]), axis='index')
   A  B
0  3  3
1  3  4
2  5  6
```

```
>>> df.clip_lower(np.array([4, 5]), axis='columns')
   A  B
0  4  5
```

(continues on next page)

(continued from previous page)

1	4	5
2	5	6

**pandas.Panel.clip\_upper**`Panel.clip_upper` (*threshold*, *axis=None*, *inplace=False*)

Return copy of input with values above given value(s) truncated.

**Parameters****threshold** [float or array\_like]**axis** : int or string axis name, optional

Align object with threshold along the given axis.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

New in version 0.21.0.

**Returns****clipped** [same type as input]**See also:**`clip`**pandas.Panel.compound**`Panel.compound` (*axis=None*, *skipna=None*, *level=None*)

Return the compound percentage of the values for the requested axis

**Parameters****axis** [{items (0), major\_axis (1), minor\_axis (2)}]**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns****compounded** [DataFrame or Panel (if level specified)]

### pandas.Panel.conform

`Panel.conform` (*frame*, *axis*='items')

Conform input DataFrame to align with chosen axis pair.

#### Parameters

**frame** [DataFrame]

**axis** : {'items', 'major', 'minor'}

Axis the input corresponds to. E.g., if *axis*='major', then the frame's columns would be items, and the index would be values of the minor axis

#### Returns

**DataFrame**

### pandas.Panel consolidate

`Panel.consolidate` (*inplace*=False)

Compute NDFrame with “consolidated” internals (data of each dtype grouped together in a single ndarray).

Deprecated since version 0.20.0: Consolidate will be an internal implementation only.

### pandas.Panel.convert\_objects

`Panel.convert_objects` (*convert\_dates*=True, *convert\_numeric*=False, *convert\_timedeltas*=True, *copy*=True)

Attempt to infer better dtype for object columns.

Deprecated since version 0.21.0.

**Parameters** **convert\_dates** : boolean, default True

If True, convert to date where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

**convert\_numeric** : boolean, default False

If True, attempt to coerce to numbers (including strings), with unconvertible values becoming NaN.

**convert\_timedeltas** : boolean, default True

If True, convert to timedelta where possible. If 'coerce', force conversion, with unconvertible values becoming NaT.

**copy** : boolean, default True

If True, return a copy even if no copy is necessary (e.g. no conversion was done).  
Note: This is meant for internal use, and should not be confused with *inplace*.

#### Returns

**converted** [same as input object]

See also:

[`pandas.to\_datetime`](#) Convert argument to datetime.

`pandas.to_timedelta` Convert argument to timedelta.

`pandas.to_numeric` Return a fixed frequency timedelta index, with day as the default.

## `pandas.Panel.copy`

`Panel.copy(deep=True)`

Make a copy of this object's indices and data.

When `deep=True` (default), a new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

When `deep=False`, a new object will be created without copying the calling object's data or index (only references to the data and index are copied). Any changes to the data of the original will be reflected in the shallow copy (and vice versa).

**Parameters** `deep` : bool, default True

Make a deep copy, including a copy of the data and the indices. With `deep=False` neither the indices nor the data are copied.

**Returns** `copy` : Series, DataFrame or Panel

Object type matches caller.

## Notes

When `deep=True`, data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to `copy.deepcopy` in the Standard Library, which recursively copies object data (see examples below).

While `Index` objects are copied when `deep=True`, the underlying numpy array is not copied for performance reasons. Since `Index` is immutable, the underlying data can be safely shared and a copy is not needed.

## Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

**Shallow copy versus default (deep) copy:**

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> deep = s.copy()
>>> shallow = s.copy(deep=False)
```

Shallow copy shares data and index with original.

```
>>> s is shallow
False
>>> s.values is shallow.values and s.index is shallow.index
True
```

Deep copy has own copy of data and index.

```
>>> s is deep
False
>>> s.values is deep.values or s.index is deep.index
False
```

Updates to the data shared by shallow copy and original is reflected in both; deep copy remains unchanged.

```
>>> s[0] = 3
>>> shallow[1] = 4
>>> s
a    3
b    4
dtype: int64
>>> shallow
a    3
b    4
dtype: int64
>>> deep
a    1
b    2
dtype: int64
```

Note that when copying an object containing Python objects, a deep copy will copy the data, but will not do so recursively. Updating a nested data object will be reflected in the deep copy.

```
>>> s = pd.Series([[1, 2], [3, 4]])
>>> deep = s.copy()
>>> s[0][0] = 10
>>> s
0    [10, 2]
1     [3, 4]
dtype: object
>>> deep
0    [10, 2]
1     [3, 4]
dtype: object
```

## pandas.Panel.count

`Panel.count` (*axis='major'*)

Return number of observations over requested axis.

### Parameters

**axis** [{‘items’, ‘major’, ‘minor’} or {0, 1, 2}]

### Returns

**count** [DataFrame]

**pandas.Panel.cummax**

`Panel.cummax` (*axis=None, skipna=True, \*args, \*\*kwargs*)

Return cumulative maximum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative maximum.

**Parameters** *axis*: {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs**:

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**

**cummax** [DataFrame or Panel]

**See also:**

[`pandas.core.window.Expanding.max`](#) Similar functionality but ignores NaN values.

[`Panel.max`](#) Return the maximum over Panel axis.

[`Panel.cummax`](#) Return cumulative maximum over Panel axis.

[`Panel.cummin`](#) Return cumulative minimum over Panel axis.

[`Panel.cumsum`](#) Return cumulative sum over Panel axis.

[`Panel.cumprod`](#) Return cumulative product over Panel axis.

**Examples****Series**

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummax()
0    2.0
1    NaN
2    5.0
3    5.0
4    5.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummax(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the maximum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummax()
   A    B
0  2.0  1.0
1  3.0  NaN
2  3.0  1.0
```

To iterate over columns and find the maximum in each row, use `axis=1`

```
>>> df.cummax(axis=1)
   A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  1.0
```

### pandas.Panel.cummin

`Panel.cummin` (*axis=None, skipna=True, \*args, \*\*kwargs*)

Return cumulative minimum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative minimum.

**Parameters** `axis`: {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs**:

Additional keywords have no effect but might be accepted for compatibility with NumPy.

**Returns**



**cummin** [DataFrame or Panel]

See also:

*pandas.core.window.Expanding.min* Similar functionality but ignores NaN values.

*Panel.min* Return the minimum over Panel axis.

*Panel.cummax* Return cumulative maximum over Panel axis.

*Panel.cummin* Return cumulative minimum over Panel axis.

*Panel.cumsum* Return cumulative sum over Panel axis.

*Panel.cumprod* Return cumulative product over Panel axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cummin()
0    2.0
1    NaN
2    2.0
3   -1.0
4   -1.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cummin(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
```

(continues on next page)

(continued from previous page)

```
1  3.0  NaN
2  1.0   0.0
```

By default, iterates over rows and finds the minimum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cummin()
      A      B
0  2.0   1.0
1  2.0   NaN
2  1.0   0.0
```

To iterate over columns and find the minimum in each row, use `axis=1`

```
>>> df.cummin(axis=1)
      A      B
0  2.0   1.0
1  3.0   NaN
2  1.0   0.0
```

## pandas.Panel.cumprod

`Panel.cumprod(axis=None, skipna=True, *args, **kwargs)`

Return cumulative product over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative product.

**Parameters** `axis`: {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna**: boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs**:

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**cumprod** [DataFrame or Panel]

See also:

**pandas.core.window.Expanding.prod** Similar functionality but ignores NaN values.

**Panel.prod** Return the product over Panel axis.

**Panel.cummax** Return cumulative maximum over Panel axis.

**Panel.cummin** Return cumulative minimum over Panel axis.

**Panel.cumsum** Return cumulative sum over Panel axis.

**Panel.cumprod** Return cumulative product over Panel axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumprod()
0    2.0
1    NaN
2   10.0
3  -10.0
4   -0.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumprod(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A    B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the product in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumprod()
   A    B
0  2.0  1.0
1  6.0  NaN
2  6.0  0.0
```

To iterate over columns and find the product in each row, use `axis=1`

```
>>> df.cumprod(axis=1)
   A    B
0  2.0  2.0
1  3.0  NaN
2  1.0  0.0
```

## pandas.Panel.cumsum

`Panel.cumsum` (*axis=None*, *skipna=True*, *\*args*, *\*\*kwargs*)

Return cumulative sum over a DataFrame or Series axis.

Returns a DataFrame or Series of the same size containing the cumulative sum.

**Parameters** *axis* : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis. 0 is equivalent to None or 'index'.

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**\*args, \*\*kwargs** :

Additional keywords have no effect but might be accepted for compatibility with NumPy.

### Returns

**cumsum** [DataFrame or Panel]

See also:

[`pandas.core.window.Expanding.sum`](#) Similar functionality but ignores NaN values.

[`Panel.sum`](#) Return the sum over Panel axis.

[`Panel.cummax`](#) Return cumulative maximum over Panel axis.

[`Panel.cummin`](#) Return cumulative minimum over Panel axis.

[`Panel.cumsum`](#) Return cumulative sum over Panel axis.

[`Panel.cumprod`](#) Return cumulative product over Panel axis.

## Examples

### Series

```
>>> s = pd.Series([2, np.nan, 5, -1, 0])
>>> s
0    2.0
1    NaN
2    5.0
3   -1.0
4    0.0
dtype: float64
```

By default, NA values are ignored.

```
>>> s.cumsum()
0    2.0
1    NaN
2    7.0
3    6.0
4    6.0
dtype: float64
```

To include NA values in the operation, use `skipna=False`

```
>>> s.cumsum(skipna=False)
0    2.0
1    NaN
2    NaN
3    NaN
4    NaN
dtype: float64
```

### DataFrame

```
>>> df = pd.DataFrame([[2.0, 1.0],
...                    [3.0, np.nan],
...                    [1.0, 0.0]],
...                    columns=list('AB'))
>>> df
   A  B
0  2.0  1.0
1  3.0  NaN
2  1.0  0.0
```

By default, iterates over rows and finds the sum in each column. This is equivalent to `axis=None` or `axis='index'`.

```
>>> df.cumsum()
   A  B
0  2.0  1.0
1  5.0  NaN
2  6.0  1.0
```

To iterate over columns and find the sum in each row, use `axis=1`

```
>>> df.cumsum(axis=1)
   A  B
0  2.0  3.0
1  3.0  NaN
2  1.0  1.0
```

### pandas.Panel.describe

`Panel.describe` (*percentiles=None, include=None, exclude=None*)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters** `percentiles` : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

**include** : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use 'category'
- None (default) : The result will include all numeric columns.

**exclude** : list-like of dtypes or None (default), optional,

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use 'category'
- None (default) : The result will exclude nothing.

## Returns

**summary**: `Series/DataFrame` of summary statistics

See also:

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

## Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as `lower`, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

## Examples

Describing a numeric Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
top        a
freq       2
dtype: object
```

Describing a timestamp Series.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count              3
unique             2
top      2010-01-01 00:00:00
freq              2
first    2000-01-01 00:00:00
last     2010-01-01 00:00:00
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({ 'object': ['a', 'b', 'c'],
...                     'numeric': [1, 2, 3],
...                     'categorical': pd.Categorical(['d', 'e', 'f'])
... })
>>> df.describe()
           numeric
count           3.0
mean            2.0
std             1.0
min             1.0
25%            1.5
50%            2.0
75%            2.5
max            3.0
```

Describing all columns of a DataFrame regardless of data type.

```
>>> df.describe(include='all')
      categorical  numeric  object
count           3        3.0      3
unique           3        NaN      3
top              f        NaN      c
freq            1        NaN      1
mean            NaN        2.0    NaN
std             NaN        1.0    NaN
min             NaN        1.0    NaN
25%             NaN        1.5    NaN
50%             NaN        2.0    NaN
75%             NaN        2.5    NaN
max             NaN        3.0    NaN
```

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```
>>> df.describe(include=[np.number])
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
      object
count      3
unique      3
top         c
freq        1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
      categorical
count           3
unique           3
```

(continues on next page)



(continued from previous page)

top	f
freq	1

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
           categorical  object
count                3      3
unique               3      3
top                 f      c
freq                1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
           categorical  numeric
count                3      3.0
unique               3      NaN
top                 f      NaN
freq                1      NaN
mean                NaN      2.0
std                 NaN      1.0
min                 NaN      1.0
25%                 NaN      1.5
50%                 NaN      2.0
75%                 NaN      2.5
max                 NaN      3.0
```

## pandas.Panel.div

`Panel.div(other, axis=0)`

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

**See also:**

`Panel.rtruediv`

## pandas.Panel.divide

`Panel.divide(other, axis=0)`

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

### Parameters

**other** [DataFrame or Panel]  
**axis** : {items, major\_axis, minor\_axis}  
Axis to broadcast over

**Returns**

**Panel**

**See also:**

`Panel.rtruediv`

## **pandas.Panel.dropna**

`Panel.dropna` (*axis=0, how='any', inplace=False*)  
Drop 2D from panel, holding passed axis constant

**Parameters** **axis** : int, default 0

Axis to hold constant. E.g. axis=1 will drop major\_axis entries having a certain amount of NA data

**how** : { 'all', 'any' }, default 'any'

'any': one or more values are NA in the DataFrame along the axis. For 'all' they all must be.

**inplace** : bool, default False

If True, do operation inplace and return None.

**Returns**

**dropped** [Panel]

## **pandas.Panel.eq**

`Panel.eq` (*other, axis=None*)  
Wrapper for comparison method eq

## **pandas.Panel.equals**

`Panel.equals` (*other*)  
Determines if two NDFrame objects contain the same elements. NaNs in the same location are considered equal.

## **pandas.Panel.ffill**

`Panel.ffill` (*axis=None, inplace=False, limit=None, downcast=None*)  
Synonym for `DataFrame.fillna(method='ffill')`

**pandas.Panel.fillna**

`Panel.fillna` (*value=None, method=None, axis=None, inplace=False, limit=None, downcast=None, \*\*kwargs*)

Fill NA/NaN values using the specified method

**Parameters** **value** : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

**method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**axis** : [{0, 1, 2, 'items', 'major\_axis', 'minor\_axis'}]

**inplace** : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

**limit** : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

**downcast** : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

**Returns**

**filled** [Panel]

**See also:**

[\*interpolate\*](#) Fill NaN values using interpolation.

[\*reindex, asfreq\*](#)

**Examples**

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
   A  B  C  D
```

(continues on next page)

(continued from previous page)

0	NaN	2.0	NaN	0
1	3.0	4.0	NaN	1
2	NaN	NaN	NaN	5
3	NaN	3.0	NaN	4

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
   A    B    C    D
0  0.0  2.0  0.0  0
1  3.0  4.0  0.0  1
2  0.0  0.0  0.0  5
3  0.0  3.0  0.0  4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
   A    B    C    D
0  NaN  2.0  NaN  0
1  3.0  4.0  NaN  1
2  3.0  4.0  NaN  5
3  3.0  3.0  NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  2.0  1
2  0.0  1.0  2.0  5
3  0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  NaN  1
2  NaN  1.0  NaN  5
3  NaN  3.0  NaN  4
```

## pandas.Panel.filter

`Panel.filter` (*items=None, like=None, regex=None, axis=None*)

Subset rows or columns of dataframe according to labels in the specified index.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

**Parameters** *items* : list-like

List of info axis to restrict to (must not all be present)

**like** : string

Keep info axis where “arg in col == True”

**regex** : string (regular expression)

Keep info axis with `re.search(regex, col) == True`

**axis** : int or string axis name

The axis to filter on. By default this is the info axis, 'index' for Series, 'columns' for DataFrame

### Returns

same type as input object

See also:

[`pandas.DataFrame.loc`](#)

### Notes

The `items`, `like`, and `regex` parameters are enforced to be mutually exclusive.

`axis` defaults to the info axis that is used when indexing with `[]`.

### Examples

```
>>> df
one  two  three
mouse    1    2    3
rabbit   4    5    6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
one  three
mouse    1    3
rabbit   4    6
```

```
>>> # select columns by regular expression
>>> df.filter(regex='e$', axis=1)
one  three
mouse    1    3
rabbit   4    6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
one  two  three
rabbit   4    5    6
```

## pandas.Panel.first

`Panel.first` (*offset*)

Convenience method for subsetting initial periods of time series data based on a date offset.

### Parameters

**offset** [string, `DateOffset`, `dateutil.relativedelta`]

**Returns****subset** [type of caller]**Raises** **TypeError**If the index is not a *DatetimeIndex***See also:***last* Select final periods of time series based on a date offset*at\_time* Select values at a particular time of the day*between\_time* Select values between particular times of the day**Examples**

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
              A
2018-04-09    1
2018-04-11    2
2018-04-13    3
2018-04-15    4
```

Get the rows for the first 3 days:

```
>>> ts.first('3D')
              A
2018-04-09    1
2018-04-11    2
```

Notice the data for 3 first calendar days were returned, not the first 3 days observed in the dataset, and therefore data for 2018-04-13 was not returned.

**pandas.Panel.first\_valid\_index****Panel.first\_valid\_index()**

Return index for first non-NA/null value.

**Returns****scalar** [type of index]**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

**pandas.Panel.floordiv****Panel.floordiv** (*other*, *axis=0*)Integer division of series and other, element-wise (binary operator *floordiv*). Equivalent to `panel // other`.

**Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

**Returns**

**Panel**

**See also:**

[`Panel.rfloordiv`](#)

**pandas.Panel.fromDict**

**classmethod** `Panel.fromDict` (*data*, *intersect=False*, *orient='items'*, *dtype=None*)

Construct Panel from dict of DataFrame objects

**Parameters** **data** : dict

{field : DataFrame}

**intersect** : boolean

Intersect indexes of input DataFrames

**orient** : {'items', 'minor'}, default 'items'

The “orientation” of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

**dtype** : dtype, default None

Data type to force, otherwise infer

**Returns**

**Panel**

**pandas.Panel.from\_dict**

**classmethod** `Panel.from_dict` (*data*, *intersect=False*, *orient='items'*, *dtype=None*)

Construct Panel from dict of DataFrame objects

**Parameters** **data** : dict

{field : DataFrame}

**intersect** : boolean

Intersect indexes of input DataFrames

**orient** : {'items', 'minor'}, default 'items'

The “orientation” of the data. If the keys of the passed dict should be the items of the result panel, pass 'items' (default). Otherwise if the columns of the values of the passed DataFrame objects should be the items (which in the case of mixed-dtype data you should do), instead pass 'minor'

**dtype** : dtype, default None

Data type to force, otherwise infer

**Returns**

**Panel**

### **pandas.Panel.ge**

`Panel.ge` (*other*, *axis=None*)

Wrapper for comparison method `ge`

### **pandas.Panel.get**

`Panel.get` (*key*, *default=None*)

Get item from object for given key (DataFrame column, Panel slice, etc.). Returns default value if not found.

**Parameters**

**key** [object]

**Returns**

**value** [type of items contained in object]

### **pandas.Panel.get\_dtype\_counts**

`Panel.get_dtype_counts` ()

Return counts of unique dtypes in this object.

**Returns dtype** : Series

Series with the count of columns with each dtype.

**See also:**

[\*dtypes\*](#) Return the dtypes in this object.

### **Examples**

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
   str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_dtype_counts()
float64    1
int64      1
object     1
dtype: int64
```



**pandas.Panel.get\_ftype\_counts****Panel.get\_ftype\_counts()**

Return counts of unique ftypes in this object.

Deprecated since version 0.23.0.

This is useful for SparseDataFrame or for DataFrames containing sparse arrays.

**Returns dtype** : Series

Series with the count of columns with each type and sparsity (dense/sparse)

**See also:****ftypes** Return ftypes (indication of sparse/dense and dtype) in this object.**Examples**

```
>>> a = [['a', 1, 1.0], ['b', 2, 2.0], ['c', 3, 3.0]]
>>> df = pd.DataFrame(a, columns=['str', 'int', 'float'])
>>> df
   str  int  float
0   a    1    1.0
1   b    2    2.0
2   c    3    3.0
```

```
>>> df.get_ftype_counts()
float64:dense    1
int64:dense      1
object:dense     1
dtype: int64
```

**pandas.Panel.get\_value****Panel.get\_value(\*args, \*\*kwargs)**

Quickly retrieve single value at (item, major, minor) location

Deprecated since version 0.21.0.

Please use .at[] or .iat[] accessors.

**Parameters****item** [item label (panel item)]**major** [major axis label (panel item row)]**minor** [minor axis label (panel item column)]**takeable** [interpret the passed labels as indexers, default False]**Returns****value** [scalar value]

## pandas.Panel.get\_values

`Panel.get_values()`

Return an ndarray after converting sparse values to dense.

This is the same as `.values` for non-sparse data. For sparse data contained in a *pandas.SparseArray*, the data are first converted to a dense representation.

**Returns** `numpy.ndarray`

Numpy representation of DataFrame

**See also:**

**values** Numpy representation of DataFrame.

**pandas.SparseArray** Container for sparse data.

## Examples

```
>>> df = pd.DataFrame({'a': [1, 2], 'b': [True, False],
...                    'c': [1.0, 2.0]})
>>> df
   a      b      c
0  1   True  1.0
1  2  False  2.0
```

```
>>> df.get_values()
array([[1, True, 1.0], [2, False, 2.0]], dtype=object)
```

```
>>> df = pd.DataFrame({"a": pd.SparseArray([1, None, None]),
...                    "c": [1.0, 2.0, 3.0]})
>>> df
   a      c
0  1.0  1.0
1  NaN  2.0
2  NaN  3.0
```

```
>>> df.get_values()
array([[ 1.,  1.],
       [nan,  2.],
       [nan,  3.]])
```

## pandas.Panel.groupby

`Panel.groupby(function, axis='major')`

Group data on given axis, returning GroupBy object

**Parameters** `function` : callable

Mapping function for chosen access

**axis** [{‘major’, ‘minor’, ‘items’}, default ‘major’]

**Returns**

**grouped** [PanelGroupBy]**pandas.Panel.gt****Panel.gt** (*other, axis=None*)

Wrapper for comparison method gt

**pandas.Panel.head****Panel.head** (*n=5*)Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

**Parameters** *n* : int, default 5

Number of rows to select.

**Returns** *obj\_head* : type of callerThe first *n* rows of the caller object.**See also:**[\*pandas.DataFrame.tail\*](#) Returns the last *n* rows.**Examples**

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0  alligator
1      bee
2   falcon
3     lion
4   monkey
5   parrot
6   shark
7   whale
8   zebra
```

Viewing the first 5 lines

```
>>> df.head()
   animal
0  alligator
1      bee
2   falcon
3     lion
4   monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
   animal
0  alligator
1      bee
2    falcon
```

## pandas.Panel.infer\_objects

`Panel.infer_objects()`

Attempt to infer better dtypes for object columns.

Attempts soft conversion of object-dtyped columns, leaving non-object and unconvertible columns unchanged. The inference rules are the same as during normal Series/DataFrame construction.

New in version 0.21.0.

### Returns

**converted** [same type as input object]

See also:

[`pandas.to\_datetime`](#) Convert argument to datetime.

[`pandas.to\_timedelta`](#) Convert argument to timedelta.

[`pandas.to\_numeric`](#) Convert argument to numeric typeR

## Examples

```
>>> df = pd.DataFrame({"A": ["a", 1, 2, 3]})
>>> df = df.iloc[1:]
>>> df
   A
1  1
2  2
3  3
```

```
>>> df.dtypes
A    object
dtype: object
```

```
>>> df.infer_objects().dtypes
A    int64
dtype: object
```

## pandas.Panel.interpolate

`Panel.interpolate` (*method='linear', axis=0, limit=None, inplace=False, limit\_direction='forward', limit\_area=None, downcast=None, \*\*kwargs*)

Interpolate values according to different methods.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

**Parameters** `method`: {'linear', 'time', 'index', 'values', 'nearest', 'zero',

‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘krogh’, ‘polynomial’, ‘spline’, ‘piecewise\_polynomial’, ‘from\_derivatives’, ‘pchip’, ‘akima’}

- ‘linear’: ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- ‘time’: interpolation works on daily and higher resolution data to interpolate given length of interval
- ‘index’, ‘values’: use the actual numerical values of the index
- ‘nearest’, ‘zero’, ‘slinear’, ‘quadratic’, ‘cubic’, ‘barycentric’, ‘polynomial’ is passed to `scipy.interpolate.interpld`. Both ‘polynomial’ and ‘spline’ require that you also specify an *order* (int), e.g. `df.interpolate(method='polynomial', order=4)`. These use the actual numerical values of the index.
- ‘krogh’, ‘piecewise\_polynomial’, ‘spline’, ‘pchip’ and ‘akima’ are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the [scipy documentation](#) and [tutorial documentation](#)
- ‘from\_derivatives’ refers to `BPoly.from_derivatives` which replaces ‘piecewise\_polynomial’ interpolation method in scipy 0.18

New in version 0.18.1: Added support for the ‘akima’ method Added interpolate method ‘from\_derivatives’ which replaces ‘piecewise\_polynomial’ in scipy 0.18; backwards-compatible with scipy < 0.18

**axis** : {0, 1}, default 0

- 0: fill column-by-column
- 1: fill row-by-row

**limit** : int, default None.

Maximum number of consecutive NaNs to fill. Must be greater than 0.

**limit\_direction** [{‘forward’, ‘backward’, ‘both’}, default ‘forward’]

**limit\_area** : {‘inside’, ‘outside’}, default None

- None: (default) no fill restriction
- ‘inside’ Only fill NaNs surrounded by valid values (interpolate).
- ‘outside’ Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

**inplace** : bool, default False

Update the NDFrame in place if possible.

**downcast** : optional, ‘infer’ or None, defaults to None

Downcast dtypes if possible.

**kwargs** [keyword arguments to pass on to the interpolating function.]

**Returns**

**Series or DataFrame of same shape interpolated at the NaNs**

**See also:**

*reindex, replace, fillna*

**Examples**

Filling in NaNs

```
>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64
```

**pandas.Panel.isna**

**Panel.isna()**

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

**Returns NDFrame**

Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

**See also:**

**NDFrame.isnull** alias of `isna`

**NDFrame.notna** boolean inverse of `isna`

**NDFrame.dropna** omit axes labels with missing values

*isna* top-level `isna`

**Examples**

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                    'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                             pd.Timestamp('1940-04-25')],
...                    'name': ['Alfred', 'Batman', ''],
...                    'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age  born      name      toy
0    5  NaT    Alfred    None
1    6  b'1939-05-27' Batman  Batmobile
2  NaN  b'1940-04-25'     ''    Joker
```

(continues on next page)

(continued from previous page)

0	5.0	NaT	Alfred	None
1	6.0	1939-05-27	Batman	Batmobile
2	NaN	1940-04-25		Joker

```
>>> df.isna()
      age  born  name  toy
0  False  True False  True
1  False False False False
2   True False False False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

## pandas.Panel.isnull

`Panel.isnull()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None` or `numpy.NaN`, gets mapped to `True` values. Everything else gets mapped to `False` values. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

### Returns NDFrame

Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

**NDFrame.isnull** alias of `isna`

**NDFrame.notna** boolean inverse of `isna`

**NDFrame.dropna** omit axes labels with missing values

[\*isna\*](#) top-level `isna`

## Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born  name      toy
0  5.0      NaT  Alfred    None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.isna()
   age  born  name  toy
0  False  True  False  True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

## pandas.Panel.iteritems

`Panel.iteritems()`

Iterate over (label, values) on info axis

This is index for Series, columns for DataFrame, `major_axis` for Panel, and so on.

## pandas.Panel.join

`Panel.join(other, how='left', lsuffix="", rsuffix="")`

Join items with other Panel either on major and minor axes column

**Parameters** `other` : Panel or list of Panels

Index should be similar to one of the columns in this one

**how** : { 'left', 'right', 'outer', 'inner' }

How to handle indexes of the two objects. Default: 'left' for joining on index,  
None otherwise \* left: use calling frame's index \* right: use input frame's index  
\* outer: form union of indexes \* inner: use intersection of indexes

**lsuffix** : string



Suffix to use from left frame's overlapping columns

**rsuffix** : string

Suffix to use from right frame's overlapping columns

### Returns

**joined** [Panel]

## pandas.Panel.keys

`Panel.keys()`

Get the 'info axis' (see Indexing for more)

This is index for Series, columns for DataFrame and major\_axis for Panel.

## pandas.Panel.kurt

`Panel.kurt(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**kurt** [DataFrame or Panel (if level specified)]

## pandas.Panel.kurtosis

`Panel.kurtosis(axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased kurtosis over requested axis using Fisher's definition of kurtosis (kurtosis of normal == 0.0). Normalized by N-1

### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**kurt** [DataFrame or Panel (if level specified)]

## pandas.Panel.last

`Panel.last` (*offset*)

Convenience method for subsetting final periods of time series data based on a date offset.

### Parameters

**offset** [string, DateOffset, dateutil.relativedelta]

### Returns

**subset** [type of caller]

### Raises

**TypeError**

If the index is not a *DatetimeIndex*

### See also:

*first* Select initial periods of time series based on a date offset

*at\_time* Select values at a particular time of the day

*between\_time* Select values between particular times of the day

## Examples

```
>>> i = pd.date_range('2018-04-09', periods=4, freq='2D')
>>> ts = pd.DataFrame({'A': [1,2,3,4]}, index=i)
>>> ts
              A
2018-04-09    1
2018-04-11    2
2018-04-13    3
2018-04-15    4
```

Get the rows for the last 3 days:

```
>>> ts.last('3D')
              A
2018-04-13    3
2018-04-15    4
```

Notice the data for 3 last calendar days were returned, not the last 3 observed days in the dataset, and therefore data for 2018-04-11 was not returned.

**pandas.Panel.last\_valid\_index**

`Panel.last_valid_index()`  
Return index for last non-NA/null value.

**Returns**

**scalar** [type of index]

**Notes**

If all elements are non-NA/null, returns None. Also returns None for empty NDFrame.

**pandas.Panel.le**

`Panel.le(other, axis=None)`  
Wrapper for comparison method `le`

**pandas.Panel.lt**

`Panel.lt(other, axis=None)`  
Wrapper for comparison method `lt`

**pandas.Panel.mad**

`Panel.mad(axis=None, skipna=None, level=None)`  
Return the mean absolute deviation of the values for the requested axis

**Parameters**

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**mad** [DataFrame or Panel (if level specified)]

**pandas.Panel.major\_xs**

`Panel.major_xs(key)`  
Return slice of panel along major axis

**Parameters** **key** : object

Major axis label

**Returns** **y** : DataFrame

index -> minor axis, columns -> items

## Notes

major\_xs is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of major\_xs functionality, see [MultiIndex Slicers](#)

## pandas.Panel.mask

`Panel.mask(cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try_cast=False, raise_on_error=None)`

Return an object of same shape as self and whose corresponding entries are from self where *cond* is False and otherwise are from *other*.

**Parameters** **cond** : boolean NDFrame, array-like, or callable

Where *cond* is False, keep the original value. Where True, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as cond.

**other** : scalar, NDFrame, or callable

Entries where *cond* is True are replaced with corresponding value from *other*. If *other* is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as other.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

**axis** [alignment axis if needed, default None]

**level** [alignment level if needed, default None]

**errors** : str, {'raise', 'ignore'}, default 'raise'

- **raise** : allow exceptions to be raised
- **ignore** : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

### Returns

**wh** [same type as caller]

### See also:

`DataFrame.where()`

### Notes

The mask method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `False` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for `DataFrame.where()` differs from `numpy.where()`. Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `mask` documentation in [indexing](#).

### Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1     1.0
2     2.0
3     3.0
4     4.0
```

```
>>> s.mask(s > 0)
0     0.0
1     NaN
2     NaN
3     NaN
4     NaN
```

```
>>> s.where(s > 1, 10)
0     10.0
1     10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
```

(continues on next page)

(continued from previous page)

```

>>> df.where(m, -df) == np.where(m, df, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True

```

### pandas.Panel.max

Panel.**max** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

**This method returns the maximum of the values in the object.** If you want the *index* of the maximum, use `idxmax`. This is the equivalent of the `numpy.ndarray` method `argmax`.

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**max** [DataFrame or Panel (if level specified)]

### pandas.Panel.mean

Panel.**mean** (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return the mean of the values for the requested axis

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**mean** [DataFrame or Panel (if level specified)]

### pandas.Panel.median

`Panel.median` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

Return the median of the values for the requested axis

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**median** [DataFrame or Panel (if level specified)]

### pandas.Panel.min

`Panel.min` (*axis=None, skipna=None, level=None, numeric\_only=None, \*\*kwargs*)

**This method returns the minimum of the values in the object.** If you want the *index* of the minimum, use `idxmin`. This is the equivalent of the `numpy.ndarray` method `argmin`.

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**min** [DataFrame or Panel (if level specified)]

## pandas.Panel.minor\_xs

`Panel.minor_xs` (*key*)

Return slice of panel along minor axis

**Parameters** **key** : object

Minor axis label

**Returns** **y** : DataFrame

index -> major axis, columns -> items

### Notes

`minor_xs` is only for getting, not setting values.

MultiIndex Slicers is a generic way to get/set values on any level or levels and is a superset of `minor_xs` functionality, see [MultiIndex Slicers](#)

## pandas.Panel.mod

`Panel.mod` (*other*, *axis=0*)

Modulo of series and other, element-wise (binary operator *mod*). Equivalent to `panel % other`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

**See also:**

[`Panel.rmod`](#)

## pandas.Panel.mul

`Panel.mul` (*other*, *axis=0*)

Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns



**Panel****See also:**`Panel.rmul`**pandas.Panel.multiply**`Panel.multiply` (*other*, *axis=0*)Multiplication of series and other, element-wise (binary operator *mul*). Equivalent to `panel * other`.**Parameters****other** [DataFrame or Panel]**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

**Returns****Panel****See also:**`Panel.rmul`**pandas.Panel.ne**`Panel.ne` (*other*, *axis=None*)Wrapper for comparison method `ne`**pandas.Panel.notna**`Panel.notna` ()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to False values.

**Returns NDFrame**

Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

**See also:****NDFrame.notnull** alias of `notna`**NDFrame.isna** boolean inverse of `notna`**NDFrame.dropna** omit axes labels with missing values**`notna`** top-level `notna`

## Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born    name    toy
0  5.0      NaT  Alfred   None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notna()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0    True
1    True
2    False
dtype: bool
```

## pandas.Panel.notnull

`Panel.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to False values.

### Returns NDFrame

Mask of bool values for each element in NDFrame that indicates whether an element is not an NA value.

See also:

**NDFrame.notnull** alias of `notna`

**NDFrame.isna** boolean inverse of `notna`

**NDFrame.dropna** omit axes labels with missing values

**notna** top-level notna

## Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame({'age': [5, 6, np.NaN],
...                     'born': [pd.NaT, pd.Timestamp('1939-05-27'),
...                               pd.Timestamp('1940-04-25')],
...                     'name': ['Alfred', 'Batman', ''],
...                     'toy': [None, 'Batmobile', 'Joker']})
>>> df
   age      born    name      toy
0  5.0      NaT  Alfred     None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25      Joker
```

```
>>> df.notna()
   age  born  name  toy
0  True False  True False
1  True  True  True  True
2 False  True  True  True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0    True
1    True
2   False
dtype: bool
```

## pandas.Panel.pct\_change

**Panel.pct\_change** (*periods=1, fill\_method='pad', limit=None, freq=None, \*\*kwargs*)

Percentage change between the current and a prior element.

Computes the percentage change from the immediately previous row by default. This is useful in comparing the percentage of change in a time series of elements.

**Parameters** **periods** : int, default 1

Periods to shift for forming percent change.

**fill\_method** : str, default 'pad'

How to handle NAs before computing percent changes.

**limit** : int, default None

The number of consecutive NAs to fill before stopping.

**freq** : DateOffset, timedelta, or offset alias string, optional

Increment to use from time series API (e.g. 'M' or BDay()).

**\*\*kwargs**

Additional keyword arguments are passed into *DataFrame.shift* or *Series.shift*.

**Returns** **chg** : Series or DataFrame

The same type as the calling object.

**See also:**

***Series.diff*** Compute the difference of two elements in a Series.

***DataFrame.diff*** Compute the difference of two elements in a DataFrame.

***Series.shift*** Shift the index by some number of periods.

***DataFrame.shift*** Shift the index by some number of periods.

## Examples

### Series

```
>>> s = pd.Series([90, 91, 85])
>>> s
0    90
1    91
2    85
dtype: int64
```

```
>>> s.pct_change()
0      NaN
1    0.011111
2   -0.065934
dtype: float64
```

```
>>> s.pct_change(periods=2)
0      NaN
1      NaN
2   -0.055556
dtype: float64
```

See the percentage change in a Series where filling NAs with last valid observation forward to next valid.

```
>>> s = pd.Series([90, 91, None, 85])
>>> s
0    90.0
1    91.0
2     NaN
3    85.0
dtype: float64
```

```
>>> s.pct_change(fill_method='ffill')
0      NaN
1    0.011111
2    0.000000
3   -0.065934
dtype: float64
```

### DataFrame

Percentage change in French franc, Deutsche Mark, and Italian lira from 1980-01-01 to 1980-03-01.

```
>>> df = pd.DataFrame({
...     'FR': [4.0405, 4.0963, 4.3149],
...     'GR': [1.7246, 1.7482, 1.8519],
...     'IT': [804.74, 810.01, 860.13]},
...     index=['1980-01-01', '1980-02-01', '1980-03-01'])
>>> df
```

	FR	GR	IT
1980-01-01	4.0405	1.7246	804.74
1980-02-01	4.0963	1.7482	810.01
1980-03-01	4.3149	1.8519	860.13

```
>>> df.pct_change()
```

	FR	GR	IT
1980-01-01	NaN	NaN	NaN
1980-02-01	0.013810	0.013684	0.006549
1980-03-01	0.053365	0.059318	0.061876

Percentage of change in GOOG and APPL stock volume. Shows computing the percentage change between columns.

```
>>> df = pd.DataFrame({
...     '2016': [1769950, 30586265],
...     '2015': [1500923, 40912316],
...     '2014': [1371819, 41403351]},
...     index=['GOOG', 'APPL'])
>>> df
```

	2016	2015	2014
GOOG	1769950	1500923	1371819
APPL	30586265	40912316	41403351

```
>>> df.pct_change(axis='columns')
```

	2016	2015	2014
GOOG	NaN	-0.151997	-0.086016
APPL	NaN	0.337604	0.012002

### pandas.Panel.pipe

`Panel.pipe(func, *args, **kwargs)`  
 Apply func(self, \*args, \*\*kwargs)

**Parameters** `func`: function

function to apply to the NDFrame. `args`, and `kwargs` are passed into `func`. Alternatively a (callable, data\_keyword) tuple where

`data_keyword` is a string indicating the keyword of `callable` that expects the `NDFrame`.

**args** : iterable, optional

positional arguments passed into `func`.

**kwargs** : mapping, optional

a dictionary of keyword arguments passed into `func`.

### Returns

**object** [the return type of `func`.]

### See also:

*`pandas.DataFrame.apply`, `pandas.DataFrame.applymap`, `pandas.Series.map`*

### Notes

Use `.pipe` when chaining together functions that expect `Series`, `DataFrames` or `GroupBy` objects. Instead of writing

```
>>> f(g(h(df), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe(f, arg2=b, arg3=c)
...   )
```

If you have a function that takes the data as (say) the second argument, pass a tuple indicating which keyword expects the data. For example, suppose `f` takes its data as `arg2`:

```
>>> (df.pipe(h)
...   .pipe(g, arg1=a)
...   .pipe((f, 'arg2'), arg1=a, arg3=c)
...   )
```

## pandas.Panel.pop

`Panel.pop(item)`

Return item and drop from frame. Raise `KeyError` if not found.

**Parameters** `item` : str

Column label to be popped

### Returns

**popped** [Series]

### Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=('name', 'class', 'max_speed'))
>>> df
   name  class  max_speed
0  falcon   bird     389.0
1  parrot   bird      24.0
2    lion  mammal     80.5
3  monkey  mammal      NaN
```

```
>>> df.pop('class')
0    bird
1    bird
2  mammal
3  mammal
Name: class, dtype: object
```

```
>>> df
   name  max_speed
0  falcon     389.0
1  parrot      24.0
2    lion     80.5
3  monkey      NaN
```

## pandas.Panel.pow

`Panel.pow(other, axis=0)`

Exponential power of series and other, element-wise (binary operator *pow*). Equivalent to `panel ** other`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

**See also:**

`Panel.rpow`

## pandas.Panel.prod

`Panel.prod(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the product of the values for the requested axis

### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

### Returns

**prod** [DataFrame or Panel (if level specified)]

### Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

### pandas.Panel.product

`Panel.product` (*axis=None*, *skipna=None*, *level=None*, *numeric\_only=None*, *min\_count=0*, *\*\*kwargs*)

Return the product of the values for the requested axis

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None



If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

### Returns

**prod** [DataFrame or Panel (if level specified)]

## Examples

By default, the product of an empty or all-NA Series is 1

```
>>> pd.Series([]).prod()
1.0
```

This can be controlled with the `min_count` parameter

```
>>> pd.Series([]).prod(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).prod()
1.0
```

```
>>> pd.Series([np.nan]).prod(min_count=1)
nan
```

## pandas.Panel.radd

`Panel.radd(other, axis=0)`

Addition of series and other, element-wise (binary operator *radd*). Equivalent to `other + panel`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

See also:

[`Panel.add`](#)

## pandas.Panel.rank

`Panel.rank` (*axis=0*, *method='average'*, *numeric\_only=None*, *na\_option='keep'*, *ascending=True*, *pct=False*)

Compute numerical data ranks (1 through n) along axis. Equal values are assigned a rank that is the average of the ranks of those values

**Parameters** **axis** : {0 or 'index', 1 or 'columns'}, default 0

index to direct ranking

**method** : {'average', 'min', 'max', 'first', 'dense'}

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**numeric\_only** : boolean, default None

Include only float, int, boolean data. Valid only for DataFrame or Panel objects

**na\_option** : {'keep', 'top', 'bottom'}

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

**ascending** : boolean, default True

False for ranks by high (1) to low (N)

**pct** : boolean, default False

Computes percentage rank of data

### Returns

**ranks** [same type as caller]

## pandas.Panel.rdiv

`Panel.rdiv` (*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

See also:

`Panel.truediv`

## pandas.Panel.reindex

`Panel.reindex(*args, **kwargs)`

Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`

**Parameters** `items, major_axis, minor_axis` : array-like, optional (should be specified using keywords)

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**method** : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

method to use for filling holes in reindexed DataFrame. Please note: this is only applicable to DataFrames/Series with a monotonically increasing/decreasing index.

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

**copy** : boolean, default True

Return a new object, even if the passed indexes are the same

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**fill\_value** : scalar, default np.NaN

Value to use for missing values. Defaults to NaN, but can be any "compatible" value

**limit** : int, default None

Maximum number of consecutive elements to forward or backward fill

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

**Returns**

**reindexed** [Panel]

## Examples

DataFrame.reindex supports two calling conventions

- (index=index\_labels, columns=column\_labels, ...)
- (labels, axis={'index', 'columns'}, ...)

We *highly* recommend using keyword arguments to clarify your intent.

Create a dataframe with some fictional data.

```
>>> index = ['Firefox', 'Chrome', 'Safari', 'IE10', 'Konqueror']
>>> df = pd.DataFrame({
...     'http_status': [200, 200, 404, 404, 301],
...     'response_time': [0.04, 0.02, 0.07, 0.08, 1.0]},
...     index=index)
>>> df
```

	http_status	response_time
Firefox	200	0.04
Chrome	200	0.02
Safari	404	0.07
IE10	404	0.08
Konqueror	301	1.00

Create a new index and reindex the dataframe. By default values in the new index that do not have corresponding records in the dataframe are assigned NaN.

```
>>> new_index= ['Safari', 'Iceweasel', 'Comodo Dragon', 'IE10',
...             'Chrome']
>>> df.reindex(new_index)
```

	http_status	response_time
Safari	404.0	0.07
Iceweasel	NaN	NaN
Comodo Dragon	NaN	NaN
IE10	404.0	0.08
Chrome	200.0	0.02

We can fill in the missing values by passing a value to the keyword `fill_value`. Because the index is not monotonically increasing or decreasing, we cannot use arguments to the keyword method to fill the NaN values.

```
>>> df.reindex(new_index, fill_value=0)
```

	http_status	response_time
Safari	404	0.07
Iceweasel	0	0.00
Comodo Dragon	0	0.00
IE10	404	0.08
Chrome	200	0.02

```
>>> df.reindex(new_index, fill_value='missing')
```

	http_status	response_time
Safari	404	0.07
Iceweasel	missing	missing
Comodo Dragon	missing	missing
IE10	404	0.08
Chrome	200	0.02

We can also reindex the columns.

```
>>> df.reindex(columns=['http_status', 'user_agent'])
      http_status  user_agent
Firefox          200         NaN
Chrome           200         NaN
Safari           404         NaN
IE10             404         NaN
Konqueror        301         NaN
```

Or we can use “axis-style” keyword arguments

```
>>> df.reindex(['http_status', 'user_agent'], axis="columns")
      http_status  user_agent
Firefox          200         NaN
Chrome           200         NaN
Safari           404         NaN
IE10             404         NaN
Konqueror        301         NaN
```

To further illustrate the filling functionality in `reindex`, we will create a dataframe with a monotonically increasing index (for example, a sequence of dates).

```
>>> date_index = pd.date_range('1/1/2010', periods=6, freq='D')
>>> df2 = pd.DataFrame({"prices": [100, 101, np.nan, 100, 89, 88]},
...                     index=date_index)
>>> df2
      prices
2010-01-01    100
2010-01-02    101
2010-01-03     NaN
2010-01-04    100
2010-01-05     89
2010-01-06     88
```

Suppose we decide to expand the dataframe to cover a wider date range.

```
>>> date_index2 = pd.date_range('12/29/2009', periods=10, freq='D')
>>> df2.reindex(date_index2)
      prices
2009-12-29     NaN
2009-12-30     NaN
2009-12-31     NaN
2010-01-01    100
2010-01-02    101
2010-01-03     NaN
2010-01-04    100
2010-01-05     89
2010-01-06     88
2010-01-07     NaN
```

The index entries that did not have a value in the original data frame (for example, ‘2009-12-29’) are by default filled with `NaN`. If desired, we can fill in the missing values using one of several options.

For example, to backpropagate the last valid value to fill the `NaN` values, pass `bfill` as an argument to the method keyword.

```
>>> df2.reindex(date_index2, method='bfill')
      prices
```

(continues on next page)

(continued from previous page)

2009-12-29	100
2009-12-30	100
2009-12-31	100
2010-01-01	100
2010-01-02	101
2010-01-03	NaN
2010-01-04	100
2010-01-05	89
2010-01-06	88
2010-01-07	NaN

Please note that the NaN value present in the original dataframe (at index value 2010-01-03) will not be filled by any of the value propagation schemes. This is because filling while reindexing does not look at dataframe values, but only compares the original and desired indexes. If you do want to fill in the NaN values present in the original dataframe, use the `fillna()` method.

See the [user guide](#) for more.

### pandas.Panel.reindex\_axis

`Panel.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=nan)`

Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index. A new object is produced unless the new index is equivalent to the current one and `copy=False`

**Parameters** `labels` : array-like

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis** : [{0, 1, 2, 'items', 'major\_axis', 'minor\_axis'}]

**method** : {None, 'backfill'/'bfill', 'pad'/'ffill', 'nearest'}, optional

Method to use for filling holes in reindexed DataFrame:

- default: don't fill gaps
- pad / ffill: propagate last valid observation forward to next valid
- backfill / bfill: use next valid observation to fill gap
- nearest: use nearest valid observations to fill gap

**copy** : boolean, default True

Return a new object, even if the passed indexes are the same

**level** : int or name

Broadcast across a level, matching Index values on the passed MultiIndex level

**limit** : int, default None

Maximum number of consecutive elements to forward or backward fill

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

### Returns

**reindexed** [Panel]

### See also:

`reindex`, `reindex_like`

### Examples

```
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

## pandas.Panel.reindex\_like

`Panel.reindex_like` (*other*, *method=None*, *copy=True*, *limit=None*, *tolerance=None*)

Return an object with matching indices to myself.

### Parameters

**other** [Object]

**method** [string or None]

**copy** [boolean, default True]

**limit** : int, default None

Maximum number of consecutive labels to fill for inexact matches.

**tolerance** : optional

Maximum distance between labels of the other object and this object for inexact matches. Can be list-like.

New in version 0.21.0: (list-like tolerance)

### Returns

**reindexed** [same as input]

### Notes

Like calling `s.reindex(index=other.index, columns=other.columns, method=...)`

## pandas.Panel.rename

`Panel.rename` (*items=None, major\_axis=None, minor\_axis=None, \*\*kwargs*)

Alter axes input function or functions. Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error. Alternatively, change `Series.name` with a scalar value (Series only).

**Parameters** `items, major_axis, minor_axis` : scalar, list-like, dict-like or function, optional

Scalar or list-like will alter the `Series.name` attribute, and raise on `DataFrame` or `Panel`. dict-like or functions are transformations to apply to that axis' values

**copy** : boolean, default True

Also copy underlying data

**inplace** : boolean, default False

Whether to return a new `Panel`. If True then value of copy is ignored.

**level** : int or level name, default None

In case of a `MultiIndex`, only rename labels in the specified level.

### Returns

**renamed** [`Panel` (new object)]

### See also:

`pandas.NDFrame.rename_axis`

### Examples

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
>>> s.rename("my_name") # scalar, changes Series.name
0    1
1    2
2    3
Name: my_name, dtype: int64
>>> s.rename(lambda x: x ** 2) # function, changes labels
0    1
1    2
4    3
dtype: int64
>>> s.rename({1: 3, 2: 5}) # mapping, changes labels
0    1
3    2
5    3
dtype: int64
```

Since `DataFrame` doesn't have a `.name` attribute, only mapping-type arguments are allowed.



```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(2)
Traceback (most recent call last):
...
TypeError: 'int' object is not callable
```

DataFrame.rename supports two calling conventions

- (index=index\_mapper, columns=columns\_mapper, ...)
- (mapper, axis={'index', 'columns'}, ...)

We *highly* recommend using keyword arguments to clarify your intent.

```
>>> df.rename(index=str, columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename(index=str, columns={"A": "a", "C": "c"})
   a  B
0  1  4
1  2  5
2  3  6
```

Using axis-style parameters

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

See the [user guide](#) for more.

## pandas.Panel.rename\_axis

Panel.rename\_axis (mapper, axis=0, copy=True, inplace=False)

Alter the name of the index or columns.

**Parameters** **mapper** : scalar, list-like, optional

Value to set as the axis name attribute.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The index or the name of the axis.

**copy** : boolean, default True

Also copy underlying data.

**inplace** : boolean, default False

Modifies the object directly, instead of creating a new Series or DataFrame.

**Returns renamed** : Series, DataFrame, or None

The same type as the caller or None if *inplace* is True.

**See also:**

[`pandas.Series.rename`](#) Alter Series index labels or name

[`pandas.DataFrame.rename`](#) Alter DataFrame index labels or name

[`pandas.Index.rename`](#) Set new names on index

## Notes

Prior to version 0.21.0, `rename_axis` could also be used to change the axis *labels* by passing a mapping or scalar. This behavior is deprecated and will be removed in a future version. Use `rename` instead.

## Examples

### Series

```
>>> s = pd.Series([1, 2, 3])
>>> s.rename_axis("foo")
foo
0    1
1    2
2    3
dtype: int64
```

### DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename_axis("foo")
      A  B
foo
0     1  4
1     2  5
2     3  6
```

```
>>> df.rename_axis("bar", axis="columns")
bar  A  B
0     1  4
1     2  5
2     3  6
```

## `pandas.Panel.replace`

`Panel.replace` (*to\_replace=None*, *value=None*, *inplace=False*, *limit=None*, *regex=False*, *method='pad'*)

Replace values given in *to\_replace* with *value*.

Values of the NDFrame are replaced with other values dynamically. This differs from updating with `.loc` or `.iloc`, which require you to specify a location to update with some value.

**Parameters** `to_replace` : str, regex, list, dict, Series, int, float, or None

How to find the values that will be replaced.

- numeric, str or regex:
  - numeric: numeric values equal to `to_replace` will be replaced with *value*
  - str: string exactly matching `to_replace` will be replaced with *value*
  - regex: regexs matching `to_replace` will be replaced with *value*
- list of str, regex, or numeric:
  - First, if `to_replace` and *value* are both lists, they **must** be the same length.
  - Second, if `regex=True` then all of the strings in **both** lists will be interpreted as regexs otherwise they will match directly. This doesn't matter much for *value* since there are only a few possible substitution regexes you can use.
  - str, regex and numeric rules apply as above.
- dict:
  - Dicts can be used to specify different replacement values for different existing values. For example, `{ 'a': 'b', 'y': 'z' }` replaces the value 'a' with 'b' and 'y' with 'z'. To use a dict in this way the *value* parameter should be *None*.
  - For a DataFrame a dict can specify that different values should be replaced in different columns. For example, `{ 'a': 1, 'b': 'z' }` looks for the value 1 in column 'a' and the value 'z' in column 'b' and replaces these values with whatever is specified in *value*. The *value* parameter should not be *None* in this case. You can treat this as a special case of passing two lists except that you are specifying the column to search in.
  - For a DataFrame nested dictionaries, e.g., `{ 'a': { 'b': np.nan } }`, are read as follows: look in column 'a' for the value 'b' and replace it with NaN. The *value* parameter should be *None* to use a nested dict in this way. You can nest regular expressions as well. Note that column names (the top-level dictionary keys in a nested dictionary) **cannot** be regular expressions.
- None:
  - This means that the *regex* argument must be a string, compiled regular expression, or list, dict, ndarray or Series of such elements. If *value* is also *None* then this **must** be a nested dictionary or Series.

See the examples section for examples of each of these.

**value** : scalar, dict, list, str, regex, default None

Value to replace any values matching `to_replace` with. For a DataFrame a dict of values can be used to specify which value to use for each column (columns not in the dict will not be filled). Regular expressions, strings and lists or dicts of such objects are also allowed.

**inplace** : boolean, default False

If True, in place. Note: this will modify any other views on this object (e.g. a column from a DataFrame). Returns the caller if this is True.

**limit** : int, default None

Maximum size gap to forward or backward fill.

**regex** : bool or same types as *to\_replace*, default False

Whether to interpret *to\_replace* and/or *value* as regular expressions. If this is True then *to\_replace* must be a string. Alternatively, this could be a regular expression or a list, dict, or array of regular expressions in which case *to\_replace* must be None.

**method** : {'pad', 'ffill', 'bfill', None}

The method to use when for replacement, when *to\_replace* is a scalar, list or tuple and *value* is None.

Changed in version 0.23.0: Added to DataFrame.

### Returns NDFrame

Object after replacement.

### Raises AssertionError

- If *regex* is not a bool and *to\_replace* is not None.

### TypeError

- If *to\_replace* is a dict and *value* is not a list, dict, ndarray, or Series
- If *to\_replace* is None and *regex* is not compilable into a regular expression or is a list, dict, ndarray, or Series.
- When replacing multiple bool or datetime64 objects and the arguments to *to\_replace* does not match the type of the value being replaced

### ValueError

- If a list or an ndarray is passed to *to\_replace* and *value* but they are not the same length.

### See also:

**NDFrame.fillna** Fill NA values

**NDFrame.where** Replace values based on boolean condition

**Series.str.replace** Simple string replacement.

### Notes

- Regex substitution is performed under the hood with `re.sub`. The rules for substitution for `re.sub` are the same.
- Regular expressions will only substitute on strings, meaning you cannot provide, for example, a regular expression matching floating point numbers and expect the columns in your frame that have a numeric dtype to be matched. However, if those floating point numbers *are* strings, then you can do this.
- This method has *a lot* of options. You are encouraged to experiment and play with this method to gain intuition about how it works.

- When dict is used as the *to\_replace* value, it is like key(s) in the dict are the *to\_replace* part and value(s) in the dict are the *value* parameter.

## Examples

### Scalar ‘to\_replace’ and ‘value’

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.replace(0, 5)
0    5
1    1
2    2
3    3
4    4
dtype: int64
```

```
>>> df = pd.DataFrame({'A': [0, 1, 2, 3, 4],
...                    'B': [5, 6, 7, 8, 9],
...                    'C': ['a', 'b', 'c', 'd', 'e']})
>>> df.replace(0, 5)
   A  B  C
0  5  5  a
1  1  6  b
2  2  7  c
3  3  8  d
4  4  9  e
```

### List-like ‘to\_replace’

```
>>> df.replace([0, 1, 2, 3], 4)
   A  B  C
0  4  5  a
1  4  6  b
2  4  7  c
3  4  8  d
4  4  9  e
```

```
>>> df.replace([0, 1, 2, 3], [4, 3, 2, 1])
   A  B  C
0  4  5  a
1  3  6  b
2  2  7  c
3  1  8  d
4  4  9  e
```

```
>>> s.replace([1, 2], method='bfill')
0    0
1    3
2    3
3    3
4    4
dtype: int64
```

### dict-like ‘to\_replace’

```
>>> df.replace({0: 10, 1: 100})
```

	A	B	C
0	10	5	a
1	100	6	b
2	2	7	c
3	3	8	d
4	4	9	e

```
>>> df.replace({'A': 0, 'B': 5}, 100)
```

	A	B	C
0	100	100	a
1	1	6	b
2	2	7	c
3	3	8	d
4	4	9	e

```
>>> df.replace({'A': {0: 100, 4: 400}})
```

	A	B	C
0	100	5	a
1	1	6	b
2	2	7	c
3	3	8	d
4	400	9	e

### Regular expression 'to\_replace'

```
>>> df = pd.DataFrame({'A': ['bat', 'foo', 'bait'],
...                    'B': ['abc', 'bar', 'xyz']})
>>> df.replace(to_replace=r'^ba.$', value='new', regex=True)
```

	A	B
0	new	abc
1	foo	new
2	bait	xyz

```
>>> df.replace({'A': r'^ba.$'}, {'A': 'new'}, regex=True)
```

	A	B
0	new	abc
1	foo	bar
2	bait	xyz

```
>>> df.replace(regex=r'^ba.$', value='new')
```

	A	B
0	new	abc
1	foo	new
2	bait	xyz

```
>>> df.replace(regex={r'^ba.$': 'new', 'foo': 'xyz'})
```

	A	B
0	new	abc
1	xyz	new
2	bait	xyz

```
>>> df.replace(regex=[r'^ba.$', 'foo'], value='new')
```

	A	B
0	new	abc

(continues on next page)

(continued from previous page)

```
1  new  new
2  bait xyz
```

Note that when replacing multiple `bool` or `datetime64` objects, the data types in the `to_replace` parameter must match the data type of the value being replaced:

```
>>> df = pd.DataFrame({'A': [True, False, True],
...                     'B': [False, True, False]})
>>> df.replace({'a string': 'new value', True: False}) # raises
Traceback (most recent call last):
...
TypeError: Cannot compare types 'ndarray(dtype=bool)' and 'str'
```

This raises a `TypeError` because one of the dict keys is not of the correct type for replacement.

Compare the behavior of `s.replace({'a': None})` and `s.replace('a', None)` to understand the peculiarities of the `to_replace` parameter:

```
>>> s = pd.Series([10, 'a', 'a', 'b', 'a'])
```

When one uses a dict as the `to_replace` value, it is like the value(s) in the dict are equal to the `value` parameter. `s.replace({'a': None})` is equivalent to `s.replace(to_replace={'a': None}, value=None, method=None)`:

```
>>> s.replace({'a': None})
0      10
1     None
2     None
3        b
4     None
dtype: object
```

When `value=None` and `to_replace` is a scalar, list or tuple, `replace` uses the `method` parameter (default 'pad') to do the replacement. So this is why the 'a' values are being replaced by 10 in rows 1 and 2 and 'b' in row 4 in this case. The command `s.replace('a', None)` is actually equivalent to `s.replace(to_replace='a', value=None, method='pad')`:

```
>>> s.replace('a', None)
0      10
1      10
2      10
3        b
4        b
dtype: object
```

## pandas.Panel.resample

`Panel.resample(rule, how=None, axis=0, fill_method=None, closed=None, label=None, convention='start', kind=None, loffset=None, limit=None, base=0, on=None, level=None)`

Convenience method for frequency conversion and resampling of time series. Object must have a datetime-like index (`DatetimeIndex`, `PeriodIndex`, or `TimedeltaIndex`), or pass datetime-like values to the `on` or `level` keyword.

**Parameters** `rule`: string

the offset string or object representing target conversion

**axis** [int, optional, default 0]

**closed** : { 'right', 'left' }

Which side of bin interval is closed. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**label** : { 'right', 'left' }

Which bin edge label to label bucket with. The default is 'left' for all frequency offsets except for 'M', 'A', 'Q', 'BM', 'BA', 'BQ', and 'W' which all have a default of 'right'.

**convention** : { 'start', 'end', 's', 'e' }

For PeriodIndex only, controls whether to use the start or end of *rule*

**kind**: {'timestamp', 'period'}, optional

Pass 'timestamp' to convert the resulting index to a `DateTimeIndex` or 'period' to convert it to a `PeriodIndex`. By default the input representation is retained.

**loffset** : timedelta

Adjust the resampled time labels

**base** : int, default 0

For frequencies that evenly subdivide 1 day, the "origin" of the aggregated intervals. For example, for '5min' frequency, base could range from 0 through 4. Defaults to 0

**on** : string, optional

For a DataFrame, column to use instead of index for resampling. Column must be datetime-like.

New in version 0.19.0.

**level** : string or int, optional

For a MultiIndex, level (name or number) to use for resampling. Level must be datetime-like.

New in version 0.19.0.

## Returns

### Resampler object

## See also:

[\*groupby\*](#) Group by mapping, function, label, or list of labels.

## Notes

See the [user guide](#) for more.

To learn more about the offset strings, please see [this link](#).



## Examples

Start by creating a series with 9 one minute timestamps.

```
>>> index = pd.date_range('1/1/2000', periods=9, freq='T')
>>> series = pd.Series(range(9), index=index)
>>> series
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
Freq: T, dtype: int64
```

Downsample the series into 3 minute bins and sum the values of the timestamps falling into a bin.

```
>>> series.resample('3T').sum()
2000-01-01 00:00:00    3
2000-01-01 00:03:00   12
2000-01-01 00:06:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but label each bin using the right edge instead of the left. Please note that the value in the bucket used as the label is not included in the bucket, which it labels. For example, in the original series the bucket 2000-01-01 00:03:00 contains the value 3, but the summed value in the resampled bucket with the label 2000-01-01 00:03:00 does not include 3 (if it did, the summed value would be 6, not 3). To include this value close the right side of the bin interval as illustrated in the example below this one.

```
>>> series.resample('3T', label='right').sum()
2000-01-01 00:03:00    3
2000-01-01 00:06:00   12
2000-01-01 00:09:00   21
Freq: 3T, dtype: int64
```

Downsample the series into 3 minute bins as above, but close the right side of the bin interval.

```
>>> series.resample('3T', label='right', closed='right').sum()
2000-01-01 00:00:00    0
2000-01-01 00:03:00    6
2000-01-01 00:06:00   15
2000-01-01 00:09:00   15
Freq: 3T, dtype: int64
```

Upsample the series into 30 second bins.

```
>>> series.resample('30S').asfreq()[0:5] #select first 5 rows
2000-01-01 00:00:00    0.0
2000-01-01 00:00:30   NaN
2000-01-01 00:01:00    1.0
2000-01-01 00:01:30   NaN
2000-01-01 00:02:00    2.0
Freq: 30S, dtype: float64
```

Upsample the series into 30 second bins and fill the NaN values using the `pad` method.

```
>>> series.resample('30S').pad()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    0
2000-01-01 00:01:00    1
2000-01-01 00:01:30    1
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Upsample the series into 30 second bins and fill the NaN values using the `bfill` method.

```
>>> series.resample('30S').bfill()[0:5]
2000-01-01 00:00:00    0
2000-01-01 00:00:30    1
2000-01-01 00:01:00    1
2000-01-01 00:01:30    2
2000-01-01 00:02:00    2
Freq: 30S, dtype: int64
```

Pass a custom function via `apply`

```
>>> def custom_resampler(array_like):
...     return np.sum(array_like)+5

>>> series.resample('3T').apply(custom_resampler)
2000-01-01 00:00:00    8
2000-01-01 00:03:00   17
2000-01-01 00:06:00   26
Freq: 3T, dtype: int64
```

For a Series with a PeriodIndex, the keyword *convention* can be used to control whether to use the start or end of *rule*.

```
>>> s = pd.Series([1, 2], index=pd.period_range('2012-01-01',
                                                freq='A',
                                                periods=2))

>>> s
2012    1
2013    2
Freq: A-DEC, dtype: int64
```

Resample by month using ‘start’ *convention*. Values are assigned to the first month of the period.

```
>>> s.resample('M', convention='start').asfreq().head()
2012-01    1.0
2012-02    NaN
2012-03    NaN
2012-04    NaN
2012-05    NaN
Freq: M, dtype: float64
```

Resample by month using ‘end’ *convention*. Values are assigned to the last month of the period.

```
>>> s.resample('M', convention='end').asfreq()
2012-12    1.0
2013-01    NaN
```

(continues on next page)

(continued from previous page)

```

2013-02    NaN
2013-03    NaN
2013-04    NaN
2013-05    NaN
2013-06    NaN
2013-07    NaN
2013-08    NaN
2013-09    NaN
2013-10    NaN
2013-11    NaN
2013-12    2.0
Freq: M, dtype: float64

```

For DataFrame objects, the keyword `on` can be used to specify the column instead of the index for resampling.

```

>>> df = pd.DataFrame(data=9*[range(4)], columns=['a', 'b', 'c', 'd'])
>>> df['time'] = pd.date_range('1/1/2000', periods=9, freq='T')
>>> df.resample('3T', on='time').sum()

```

	a	b	c	d
time				
2000-01-01 00:00:00	0	3	6	9
2000-01-01 00:03:00	0	3	6	9
2000-01-01 00:06:00	0	3	6	9

For a DataFrame with MultiIndex, the keyword `level` can be used to specify on level the resampling needs to take place.

```

>>> time = pd.date_range('1/1/2000', periods=5, freq='T')
>>> df2 = pd.DataFrame(data=10*[range(4)],
                        columns=['a', 'b', 'c', 'd'],
                        index=pd.MultiIndex.from_product([time, [1, 2]]))
>>> df2.resample('3T', level=0).sum()

```

	a	b	c	d
2000-01-01 00:00:00	0	6	12	18
2000-01-01 00:03:00	0	4	8	12

## pandas.Panel.rfloordiv

`Panel.rfloordiv` (*other*, *axis=0*)

Integer division of series and other, element-wise (binary operator *rfloordiv*). Equivalent to `other // panel`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

See also:

*Panel.floordiv*

## **pandas.Panel.rmod**

`Panel.rmod(other, axis=0)`

Modulo of series and other, element-wise (binary operator *rmod*). Equivalent to `other % panel`.

### **Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### **Returns**

**Panel**

**See also:**

*Panel.mod*

## **pandas.Panel.rmul**

`Panel.rmul(other, axis=0)`

Multiplication of series and other, element-wise (binary operator *rmul*). Equivalent to `other * panel`.

### **Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### **Returns**

**Panel**

**See also:**

*Panel.mul*

## **pandas.Panel.round**

`Panel.round(decimals=0, *args, **kwargs)`

Round each value in Panel to a specified number of decimal places.

New in version 0.18.0.

### **Parameters** **decimals** : int

Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to the left of the decimal point.

### **Returns**

**Panel object**

See also:

`numpy.around`

### **pandas.Panel.rpow**

`Panel.rpow` (*other*, *axis=0*)

Exponential power of series and other, element-wise (binary operator *rpow*). Equivalent to `other ** panel`.

#### **Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

#### **Returns**

**Panel**

See also:

`Panel.pow`

### **pandas.Panel.rsub**

`Panel.rsub` (*other*, *axis=0*)

Subtraction of series and other, element-wise (binary operator *rsub*). Equivalent to `other - panel`.

#### **Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

#### **Returns**

**Panel**

See also:

`Panel.sub`

### **pandas.Panel.rtruediv**

`Panel.rtruediv` (*other*, *axis=0*)

Floating division of series and other, element-wise (binary operator *rtruediv*). Equivalent to `other / panel`.

#### **Parameters**

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

#### **Returns**

## Panel

See also:

`Panel.truediv`

### pandas.Panel.sample

`Panel.sample` (*n=None*, *frac=None*, *replace=False*, *weights=None*, *random\_state=None*,  
*axis=None*)

Return a random sample of items from an axis of object.

You can use *random\_state* for reproducibility.

**Parameters** *n* : int, optional

Number of items from axis to return. Cannot be used with *frac*. Default = 1 if *frac* = None.

**frac** : float, optional

Fraction of axis items to return. Cannot be used with *n*.

**replace** : boolean, optional

Sample with or without replacement. Default = False.

**weights** : str or ndarray-like, optional

Default 'None' results in equal probability weighting. If passed a Series, will align with target object on index. Index values in weights not found in sampled object will be ignored and index values in sampled object not in weights will be assigned weights of zero. If called on a DataFrame, will accept the name of a column when *axis* = 0. Unless weights are a Series, weights must be same length as axis being sampled. If weights do not sum to 1, they will be normalized to sum to 1. Missing values in the weights column will be treated as zero. inf and -inf values not allowed.

**random\_state** : int or numpy.random.RandomState, optional

Seed for the random number generator (if int), or numpy RandomState object.

**axis** : int or string, optional

Axis to sample. Accepts axis number or name. Default is stat axis for given data type (0 for Series and DataFrames, 1 for Panels).

### Returns

**A new object of same type as caller.**

### Examples

Generate an example Series and DataFrame:

```
>>> s = pd.Series(np.random.randn(50))
>>> s.head()
0    -0.038497
1     1.820773
2    -0.972766
3    -1.598270
```

(continues on next page)

(continued from previous page)

```

4    -1.095526
dtype: float64
>>> df = pd.DataFrame(np.random.randn(50, 4), columns=list('ABCD'))
>>> df.head()
   A         B         C         D
0  0.016443 -2.318952 -0.566372 -1.028078
1 -1.051921  0.438836  0.658280 -0.175797
2 -1.243569 -0.364626 -0.215065  0.057736
3  1.768216  0.404512 -0.385604 -1.457834
4  1.072446 -1.137172  0.314194 -0.046661

```

Next extract a random sample from both of these objects...

3 random elements from the Series:

```

>>> s.sample(n=3)
27    -0.994689
55    -1.049016
67    -0.224565
dtype: float64

```

And a random 10% of the DataFrame with replacement:

```

>>> df.sample(frac=0.1, replace=True)
   A         B         C         D
35  1.981780  0.142106  1.817165 -0.290805
49 -1.336199 -0.448634 -0.789640  0.217116
40  0.823173 -0.078816  1.009536  1.015108
15  1.421154 -0.055301 -1.922594 -0.019696
6   -0.148339  0.832938  1.787600 -1.383767

```

You can use *random state* for reproducibility:

```

>>> df.sample(random_state=1)
   A         B         C         D
37 -2.027662  0.103611  0.237496 -0.165867
43 -0.259323 -0.583426  1.516140 -0.479118
12 -1.686325 -0.579510  0.985195 -0.460286
8   1.167946  0.429082  1.215742 -1.636041
9   1.197475 -0.864188  1.554031 -1.505264

```

## pandas.Panel.select

`Panel.select (crit, axis=0)`

Return data corresponding to axis labels matching criteria

Deprecated since version 0.21.0: Use `df.loc[df.index.map(crit)]` to select via labels

**Parameters** `crit`: function

To be called on each index (label). Should return True or False

**axis** [int]

**Returns**

**selection** [type of caller]

## pandas.Panel.sem

`Panel.sem` (*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric\_only=None*, *\*\*kwargs*)

Return unbiased standard error of the mean over requested axis.

Normalized by N-1 by default. This can be changed using the *ddof* argument

### Parameters

**axis** : [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

### Returns

**sem** [DataFrame or Panel (if level specified)]

## pandas.Panel.set\_axis

`Panel.set_axis` (*labels*, *axis=0*, *inplace=None*)

Assign desired index to given axis.

Indexes for column or row labels can be changed by assigning a list-like or Index.

Changed in version 0.21.0: The signature is now *labels* and *axis*, consistent with the rest of pandas API. Previously, the *axis* and *labels* arguments were respectively the first and second positional arguments.

**Parameters** **labels** : list-like, Index

The values for the new index.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

The axis to update. The value 0 identifies the rows, and 1 identifies the columns.

**inplace** : boolean, default None

Whether to return a new %(klass)s instance.

**Warning:** `inplace=None` currently falls back to `True`, but in a future version, will default to `False`. Use `inplace=True` explicitly rather than relying on the default.

**Returns** **renamed** : %(klass)s or None



An object of same type as caller if inplace=False, None otherwise.

**See also:**

`pandas.DataFrame.rename_axis` Alter the name of the index or columns.

## Examples

### Series

```
>>> s = pd.Series([1, 2, 3])
>>> s
0    1
1    2
2    3
dtype: int64
```

```
>>> s.set_axis(['a', 'b', 'c'], axis=0, inplace=False)
a    1
b    2
c    3
dtype: int64
```

The original object is not modified.

```
>>> s
0    1
1    2
2    3
dtype: int64
```

### DataFrame

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
```

Change the row labels.

```
>>> df.set_axis(['a', 'b', 'c'], axis='index', inplace=False)
  A  B
a  1  4
b  2  5
c  3  6
```

Change the column labels.

```
>>> df.set_axis(['I', 'II'], axis='columns', inplace=False)
  I  II
0  1   4
1  2   5
2  3   6
```

Now, update the labels inplace.

```
>>> df.set_axis(['i', 'ii'], axis='columns', inplace=True)
>>> df
  i  ii
```

(continues on next page)

(continued from previous page)

0	1	4
1	2	5
2	3	6

### **pandas.Panel.set\_value**

`Panel.set_value (*args, **kwargs)`

Quickly set single value at (item, major, minor) location

Deprecated since version 0.21.0.

Please use `.at[]` or `.iat[]` accessors.

#### **Parameters**

**item** [item label (panel item)]

**major** [major axis label (panel item row)]

**minor** [minor axis label (panel item column)]

**value** [scalar]

**takeable** [interpret the passed labels as indexers, default False]

**Returns** `panel` : Panel

If label combo is contained, will be reference to calling Panel, otherwise a new object

### **pandas.Panel.shift**

`Panel.shift (periods=1, freq=None, axis='major')`

Shift index by desired number of periods with an optional time freq. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original. This is different from the behavior of `DataFrame.shift()`

**Parameters** `periods` : int

Number of periods to move, can be positive or negative

**freq** [DateOffset, timedelta, or time rule string, optional]

**axis** [{ 'items', 'major', 'minor' } or {0, 1, 2}]

**Returns**

**shifted** [Panel]

### **pandas.Panel.skew**

`Panel.skew (axis=None, skipna=None, level=None, numeric_only=None, **kwargs)`

Return unbiased skew over requested axis Normalized by N-1

**Parameters**

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**skew** [DataFrame or Panel (if level specified)]

### pandas.Panel.slice\_shift

Panel.**slice\_shift** (*periods=1, axis=0*)

Equivalent to *shift* without copying data. The shifted data will not include the dropped periods and the shifted axis will be smaller than the original.

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

#### Returns

**shifted** [same type as caller]

#### Notes

While the *slice\_shift* is faster than *shift*, you may pay for it later during alignment.

### pandas.Panel.sort\_index

Panel.**sort\_index** (*axis=0, level=None, ascending=True, inplace=False, kind='quicksort', na\_position='last', sort\_remaining=True*)

Sort object by labels (along an axis)

#### Parameters

**axis** [axes to direct sorting]

**level** : int or level name or list of ints or list of level names

if not None, sort on values in specified index level(s)

**ascending** : boolean, default True

Sort ascending vs. descending

**inplace** : bool, default False

if True, perform operation in-place

**kind** : { 'quicksort', 'mergesort', 'heapsort' }, default 'quicksort'

Choice of sorting algorithm. See also `ndarray.sort` for more information. *mergesort* is the only stable algorithm. For DataFrames, this option is only applied when sorting on a single column or label.

**na\_position** : { 'first', 'last' }, default 'last'

*first* puts NaNs at the beginning, *last* puts NaNs at the end. Not implemented for MultiIndex.

**sort\_remaining** : bool, default True

if true and sorting by level and index is multilevel, sort by other levels too (in order) after sorting by specified level

#### Returns

**sorted\_obj** [NDFrame]

### pandas.Panel.sort\_values

`Panel.sort_values` (*by=None*, *axis=0*, *ascending=True*, *inplace=False*, *kind='quicksort'*, *na\_position='last'*)

NOT IMPLEMENTED: do not call this method, as sorting values is not supported for Panel objects and will raise an error.

### pandas.Panel.squeeze

`Panel.squeeze` (*axis=None*)  
Squeeze length 1 dimensions.

**Parameters** **axis** : None, integer or string axis name, optional

The axis to squeeze if 1-sized.

New in version 0.20.0.

#### Returns

scalar if 1-sized, else original object

### pandas.Panel.std

`Panel.std` (*axis=None*, *skipna=None*, *level=None*, *ddof=1*, *numeric\_only=None*, *\*\*kwargs*)  
Return sample standard deviation over requested axis.

Normalized by N-1 by default. This can be changed using the *ddof* argument

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**std** [DataFrame or Panel (if level specified)]

### pandas.Panel.sub

`Panel.sub(other, axis=0)`

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

#### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

#### Returns

**Panel**

**See also:**

[`Panel.rsub`](#)

### pandas.Panel.subtract

`Panel.subtract(other, axis=0)`

Subtraction of series and other, element-wise (binary operator *sub*). Equivalent to `panel - other`.

#### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

#### Returns

**Panel**

**See also:**

[`Panel.rsub`](#)

### pandas.Panel.sum

`Panel.sum(axis=None, skipna=None, level=None, numeric_only=None, min_count=0, **kwargs)`

Return the sum of the values for the requested axis

#### Parameters

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**min\_count** : int, default 0

The required number of valid values to perform the operation. If fewer than `min_count` non-NA values are present the result will be NA.

New in version 0.22.0: Added with the default being 0. This means the sum of an all-NA or empty Series is 0, and the product of an all-NA or empty Series is 1.

### Returns

**sum** [DataFrame or Panel (if level specified)]

### Examples

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([]).sum() # min_count=0 is the default
0.0
```

This can be controlled with the `min_count` parameter. For example, if you'd like the sum of an empty series to be NaN, pass `min_count=1`.

```
>>> pd.Series([]).sum(min_count=1)
nan
```

Thanks to the `skipna` parameter, `min_count` handles all-NA and empty series identically.

```
>>> pd.Series([np.nan]).sum()
0.0
```

```
>>> pd.Series([np.nan]).sum(min_count=1)
nan
```

### pandas.Panel.swapaxes

`Panel.swapaxes` (*axis1*, *axis2*, *copy=True*)

Interchange axes and swap values axes appropriately

### Returns

**y** [same as input]

**pandas.Panel.swaplevel**`Panel.swaplevel (i=-2, j=-1, axis=0)`Swap levels *i* and *j* in a MultiIndex on a particular axis**Parameters** *i, j* : int, string (can be mixed)

Level of index to be swapped. Can pass level name as string.

**Returns****swapped** [type of caller (new object)]**.. versionchanged:: 0.18.1**The indexes *i* and *j* are now optional, and default to the two innermost levels of the index.**pandas.Panel.tail**`Panel.tail (n=5)`Return the last *n* rows.This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.**Parameters** *n* : int, default 5

Number of rows to select.

**Returns** type of callerThe last *n* rows of the caller object.**See also:**`pandas.DataFrame.head` The first *n* rows of the caller object.**Examples**

```

>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                               'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
   animal
0 alligator
1      bee
2   falcon
3     lion
4   monkey
5   parrot
6    shark
7    whale
8    zebra

```

Viewing the last 5 lines

```
>>> df.tail()
  animal
4  monkey
5  parrot
6   shark
7   whale
8   zebra
```

Viewing the last  $n$  lines (three in this case)

```
>>> df.tail(3)
  animal
6  shark
7  whale
8  zebra
```

## pandas.Panel.take

`Panel.take` (*indices*, *axis=0*, *convert=None*, *is\_copy=True*, *\*\*kwargs*)

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters** *indices* : array-like

An array of ints indicating which positions to take.

**axis** : {0 or 'index', 1 or 'columns', None}, default 0

The axis on which to select elements. 0 means that we are selecting rows, 1 means that we are selecting columns.

**convert** : bool, default True

Whether to convert negative indices into positive ones. For example,  $-1$  would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

**is\_copy** : bool, default True

Whether to return a copy of the original object or not.

**\*\*kwargs**

For compatibility with `numpy.take()`. Has no effect on the output.

**Returns** *taken* : type of caller

An array-like containing the elements taken from the object.

**See also:**

[`DataFrame.loc`](#) Select a subset of a DataFrame by labels.

[`DataFrame.iloc`](#) Select a subset of a DataFrame by positions.

[`numpy.take`](#) Take elements from an array along an axis.



## Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=['name', 'class', 'max_speed'],
...                     index=[0, 2, 3, 1])
>>> df
   name  class  max_speed
0  falcon   bird    389.0
2  parrot   bird     24.0
3   lion  mammal     80.5
1  monkey  mammal      NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
   name  class  max_speed
0  falcon   bird    389.0
1  monkey  mammal      NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
   class  max_speed
0   bird    389.0
2   bird     24.0
3  mammal     80.5
1  mammal      NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
   name  class  max_speed
1  monkey  mammal      NaN
3   lion  mammal     80.5
```

## pandas.Panel.to\_clipboard

`Panel.to_clipboard(excel=True, sep=None, **kwargs)`

Copy object to the system clipboard.

Write a text representation of object to the system clipboard. This can be pasted into Excel, for example.

**Parameters** `excel` : bool, default True

- True, use the provided separator, writing in a csv format for allowing easy pasting into excel.
- False, write a string representation of the object to the clipboard.

`sep` : str, default '\t'

Field delimiter.

**\*\*kwargs**

These parameters will be passed to `DataFrame.to_csv`.

See also:

`DataFrame.to_csv` Write a `DataFrame` to a comma-separated values (csv) file.

`read_clipboard` Read text from clipboard and pass to `read_table`.

## Notes

Requirements for your platform.

- Linux : `xclip`, or `xsel` (with `gtk` or `PyQt4` modules)
- Windows : none
- OS X : none

## Examples

Copy the contents of a `DataFrame` to the clipboard.

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6]], columns=['A', 'B', 'C'])
>>> df.to_clipboard(sep=',')
... # Wrote the following to the system clipboard:
... # ,A,B,C
... # 0,1,2,3
... # 1,4,5,6
```

We can omit the the index by passing the keyword `index` and setting it to false.

```
>>> df.to_clipboard(sep=',', index=False)
... # Wrote the following to the system clipboard:
... # A,B,C
... # 1,2,3
... # 4,5,6
```

## pandas.Panel.to\_dense

`Panel.to_dense()`

Return dense representation of `NDFrame` (as opposed to sparse)

## pandas.Panel.to\_excel

`Panel.to_excel(path, na_rep="", engine=None, **kwargs)`

Write each `DataFrame` in `Panel` to a separate excel sheet

**Parameters** `path` : string or `ExcelWriter` object

File path or existing `ExcelWriter`

`na_rep` : string, default ''

Missing data representation

**engine** : string, default None

write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**Other Parameters float\_format** : string, default None

Format string for floating point numbers

**cols** : sequence, optional

Columns to write

**header** : boolean or list of string, default True

Write out column names. If a list of string is given it is assumed to be aliases for the column names

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**startrow** [upper left cell row to dump data frame]

**startcol** [upper left cell column to dump data frame]

## Notes

Keyword arguments (and `na_rep`) are passed to the `to_excel` method for each DataFrame written.

## pandas.Panel.to\_frame

`Panel.to_frame (filter_observations=True)`

Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.

**Parameters filter\_observations** : boolean, default True

Drop (major, minor) pairs without a complete set of observations across all the items

**Returns**

y [DataFrame]

## pandas.Panel.to\_hdf

`Panel.to_hdf (path_or_buf, key, **kwargs)`

Write the contained data to an HDF5 file using HDFStore.

Hierarchical Data Format (HDF) is self-describing, allowing an application to interpret the structure and contents of a file with no outside information. One HDF file can hold a mix of related objects which can be accessed as a group or as individual objects.

In order to add another DataFrame or Series to an existing HDF file please use append mode and a different a key.

For more information see the [user guide](#).

**Parameters** **path\_or\_buf** : str or pandas.HDFStore

File path or HDFStore object.

**key** : str

Identifier for the group in the store.

**mode** : { 'a', 'w', 'r+' }, default 'a'

Mode to open file:

- 'w': write, a new file is created (an existing file with the same name would be deleted).
- 'a': append, an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+': similar to 'a', but the file must already exist.

**format** : { 'fixed', 'table' }, default 'fixed'

Possible values:

- 'fixed': Fixed format. Fast writing/reading. Not-appendable, nor searchable.
- 'table': Table format. Write as a PyTables Table structure which may perform worse but allow more flexible operations like searching / selecting subsets of the data.

**append** : bool, default False

For Table formats, append the input data to the existing.

**data\_columns** : list of columns or True, optional

List of columns to create as indexed data columns for on-disk queries, or True to use all columns. By default only the axes of the object are indexed. See [Query via Data Columns](#). Applicable only to format='table'.

**complevel** : {0-9}, optional

Specifies a compression level for data. A value of 0 disables compression.

**complib** : { 'zlib', 'lzo', 'bzip2', 'blosc' }, default 'zlib'

Specifies the compression library to be used. As of v0.20.2 these additional compressors for Blosc are supported (default if no compressor specified: 'blosc:blosclz'): { 'blosc:blosclz', 'blosc:lz4', 'blosc:lz4hc', 'blosc:snappy', 'blosc:zlib', 'blosc:zstd' }. Specifying a compression library which is not available issues a ValueError.

**fletcher32** : bool, default False

If applying compression use the fletcher32 checksum.

**dropna** : bool, default False

If true, ALL nan rows will not be written to store.

**errors** : str, default 'strict'

Specifies how encoding and decoding errors are to be handled. See the errors argument for `open()` for a full list of options.

See also:

**DataFrame.read\_hdf** Read from HDF file.

**DataFrame.to\_parquet** Write a DataFrame to the binary parquet format.

**DataFrame.to\_sql** Write to a sql table.

**DataFrame.to\_feather** Write out feather-format for DataFrames.

**DataFrame.to\_csv** Write out to a csv file.

## Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]},
...                    index=['a', 'b', 'c'])
>>> df.to_hdf('data.h5', key='df', mode='w')
```

We can add another object to the same file:

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.to_hdf('data.h5', key='s')
```

Reading from HDF file:

```
>>> pd.read_hdf('data.h5', 'df')
A  B
a  1  4
b  2  5
c  3  6
>>> pd.read_hdf('data.h5', 's')
0    1
1    2
2    3
3    4
dtype: int64
```

Deleting file with data:

```
>>> import os
>>> os.remove('data.h5')
```

## pandas.Panel.to\_json

**Panel.to\_json**(*path\_or\_buf=None, orient=None, date\_format=None, double\_precision=10, force\_ascii=True, date\_unit='ms', default\_handler=None, lines=False, compression=None, index=True*)

Convert the object to a JSON string.

Note NaN's and None will be converted to null and datetime objects will be converted to UNIX timestamps.

**Parameters** **path\_or\_buf** : string or file handle, optional

File path or object. If not specified, the result is returned as a string.

**orient** : string

Indication of expected JSON string format.

- Series
  - default is 'index'
  - allowed values are: {'split','records','index'}
- DataFrame
  - default is 'columns'
  - allowed values are: {'split','records','index','columns','values'}
- The format of the JSON string
  - 'split' : dict like {'index' -> [index], 'columns' -> [columns], 'data' -> [values]}
  - 'records' : list like [{column -> value}, ... , {column -> value}]
  - 'index' : dict like {index -> {column -> value}}
  - 'columns' : dict like {column -> {index -> value}}
  - 'values' : just the values array
  - 'table' : dict like {'schema': {schema}, 'data': {data}} describing the data, and the data component is like `orient='records'`.

Changed in version 0.20.0.

**date\_format** : {None, 'epoch', 'iso'}

Type of date conversion. 'epoch' = epoch milliseconds, 'iso' = ISO8601. The default depends on the *orient*. For `orient='table'`, the default is 'iso'. For all other orients, the default is 'epoch'.

**double\_precision** : int, default 10

The number of decimal places to use when encoding floating point values.

**force\_ascii** : boolean, default True

Force encoded string to be ASCII.

**date\_unit** : string, default 'ms' (milliseconds)

The time unit to encode to, governs timestamp and ISO8601 precision. One of 's', 'ms', 'us', 'ns' for second, millisecond, microsecond, and nanosecond respectively.

**default\_handler** : callable, default None

Handler to call if object cannot otherwise be converted to a suitable format for JSON. Should receive a single argument which is the object to convert and return a serialisable object.

**lines** : boolean, default False

If 'orient' is 'records' write out line delimited json format. Will throw ValueError if incorrect 'orient' since others are not list like.

New in version 0.19.0.

**compression** : {None, 'gzip', 'bz2', 'zip', 'xz'}

A string representing the compression to use in the output file, only used when the first argument is a filename.

New in version 0.21.0.

**index** : boolean, default True

Whether to include the index values in the JSON string. Not including the index (index=False) is only supported when orient is 'split' or 'table'.

New in version 0.23.0.

**See also:**

[`pandas.read\_json`](#)

## Examples

```
>>> df = pd.DataFrame([[ 'a', 'b'], [ 'c', 'd']],
...                    index=[ 'row 1', 'row 2'],
...                    columns=[ 'col 1', 'col 2'])
>>> df.to_json(orient='split')
'{"columns": ["col 1", "col 2"],
  "index": ["row 1", "row 2"],
  "data": [{"a", "b"}, {"c", "d"}]}'
```

Encoding/decoding a Dataframe using 'records' formatted JSON. Note that index labels are not preserved with this encoding.

```
>>> df.to_json(orient='records')
'[{"col 1": "a", "col 2": "b"}, {"col 1": "c", "col 2": "d"}]'
```

Encoding/decoding a Dataframe using 'index' formatted JSON:

```
>>> df.to_json(orient='index')
'{"row 1": {"col 1": "a", "col 2": "b"}, "row 2": {"col 1": "c", "col 2": "d"}}'
```

Encoding/decoding a Dataframe using 'columns' formatted JSON:

```
>>> df.to_json(orient='columns')
'{"col 1": {"row 1": "a", "row 2": "c"}, "col 2": {"row 1": "b", "row 2": "d"}}'
```

Encoding/decoding a Dataframe using 'values' formatted JSON:

```
>>> df.to_json(orient='values')
'[["a", "b"], ["c", "d"]]'
```

Encoding with Table Schema

```
>>> df.to_json(orient='table')
'{"schema": {"fields": [{"name": "index", "type": "string"},
                        {"name": "col 1", "type": "string"},
```

(continues on next page)

(continued from previous page)

```

        {"name": "col 2", "type": "string"}],
        "primaryKey": "index",
        "pandas_version": "0.20.0"},
    "data": [{"index": "row 1", "col 1": "a", "col 2": "b"},
              {"index": "row 2", "col 1": "c", "col 2": "d"}]}'

```

## pandas.Panel.to\_latex

`Panel.to_latex` (*buf=None, columns=None, col\_space=None, header=True, index=True, na\_rep='NaN', formatters=None, float\_format=None, sparsify=None, index\_names=True, bold\_rows=False, column\_format=None, longtable=None, escape=None, encoding=None, decimal='.', multicolumn=None, multirow=None*)

Render an object to a tabular environment table. You can splice this into a LaTeX document. Requires `\usepackage{booktabs}`.

Changed in version 0.20.2: Added to Series

*to\_latex*-specific options:

**bold\_rows** [boolean, default False] Make the row labels bold in the output

**column\_format** [str, default None] The columns format as specified in [LaTeX table format](#) e.g. 'rcl' for 3 columns

**longtable** [boolean, default will be read from the pandas config module] Default: False. Use a longtable environment instead of tabular. Requires adding a `\usepackage{longtable}` to your LaTeX preamble.

**escape** [boolean, default will be read from the pandas config module] Default: True. When set to False prevents from escaping latex special characters in column names.

**encoding** [str, default None] A string representing the encoding to use in the output file, defaults to 'ascii' on Python 2 and 'utf-8' on Python 3.

**decimal** [string, default '.'] Character recognized as decimal separator, e.g. ',' in Europe.

New in version 0.18.0.

**multicolumn** [boolean, default True] Use multicolumn to enhance MultiIndex columns. The default will be read from the config module.

New in version 0.20.0.

**multicolumn\_format** [str, default 'l'] The alignment for multicolumns, similar to *column\_format* The default will be read from the config module.

New in version 0.20.0.

**multirow** [boolean, default False] Use multirow to enhance MultiIndex rows. Requires adding a `\usepackage{multirow}` to your LaTeX preamble. Will print centered labels (instead of top-aligned) across the contained rows, separating groups via clines. The default will be read from the pandas config module.

New in version 0.20.0.



**pandas.Panel.to\_msgpack**

`Panel.to_msgpack` (*path\_or\_buf=None, encoding='utf-8', \*\*kwargs*)  
 msgpack (serialize) object to input file path

THIS IS AN EXPERIMENTAL LIBRARY and the storage format may not be stable until a future release.

**Parameters** **path** : string File path, buffer-like, or None

if None, return generated string

**append** : boolean whether to append to an existing msgpack

(default is False)

**compress** : type of compressor (zlib or blosc), default to None (no  
 compression)

**pandas.Panel.to\_pickle**

`Panel.to_pickle` (*path, compression='infer', protocol=4*)  
 Pickle (serialize) object to file.

**Parameters** **path** : str

File path where the pickled object will be stored.

**compression** : {'infer', 'gzip', 'bz2', 'zip', 'xz', None}, default 'infer'

A string representing the compression to use in the output file. By default, infers from the file extension in specified path.

New in version 0.20.0.

**protocol** : int

Int which indicates which protocol should be used by the pickler, default HIGHEST\_PROTOCOL (see [R21] paragraph 12.1.2). The possible values for this parameter depend on the version of Python. For Python 2.x, possible values are 0, 1, 2. For Python >= 3.0, 3 is a valid value. For Python >= 3.4, 4 is a valid value. A negative value for the protocol parameter is equivalent to setting its value to HIGHEST\_PROTOCOL.

New in version 0.21.0.

**See also:**

**`read_pickle`** Load pickled pandas object (or any object) from file.

**`DataFrame.to_hdf`** Write DataFrame to an HDF5 file.

**`DataFrame.to_sql`** Write DataFrame to a SQL database.

**`DataFrame.to_parquet`** Write a DataFrame to the binary parquet format.

**Examples**

```
>>> original_df = pd.DataFrame({"foo": range(5), "bar": range(5, 10)})
>>> original_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
>>> original_df.to_pickle("./dummy.pkl")
```

```
>>> unpickled_df = pd.read_pickle("./dummy.pkl")
>>> unpickled_df
   foo  bar
0    0    5
1    1    6
2    2    7
3    3    8
4    4    9
```

```
>>> import os
>>> os.remove("./dummy.pkl")
```

## pandas.Panel.to\_sparse

`Panel.to_sparse(*args, **kwargs)`

NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.

Convert to SparsePanel

## pandas.Panel.to\_sql

`Panel.to_sql(name, con, schema=None, if_exists='fail', index=True, index_label=None, chunk-size=None, dtype=None)`

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy [\[R22\]](#) are supported. Tables can be newly created, appended to, or overwritten.

**Parameters** **name** : string

Name of SQL table.

**con** : sqlalchemy.engine.Engine or sqlite3.Connection

Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects.

**schema** : string, optional

Specify the schema (if database flavor supports this). If None, use default schema.

**if\_exists** : { 'fail', 'replace', 'append' }, default 'fail'

How to behave if the table already exists.

- fail: Raise a ValueError.

- `replace`: Drop the table before inserting new values.
- `append`: Insert new values to the existing table.

**index** : boolean, default True

Write DataFrame index as a column. Uses *index\_label* as the column name in the table.

**index\_label** : string or sequence, default None

Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**chunksize** : int, optional

Rows will be written in batches of this size at a time. By default, all rows will be written at once.

**dtype** : dict, optional

Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.

**Raises** **ValueError**

When the table already exists and *if\_exists* is 'fail' (the default).

**See also:**

[`pandas.read\_sql`](#) read a DataFrame from a table

## References

[R22], [R23]

## Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
   name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just df1.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...           index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
   A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...          dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

## pandas.Panel.to\_xarray

`Panel.to_xarray()`

Return an xarray object from the pandas object.

### Returns

- a **DataArray** for a **Series**
- a **Dataset** for a **DataFrame**
- a **DataArray** for higher dims

### Notes

See the [xarray docs](#)

### Examples

```
>>> df = pd.DataFrame({'A' : [1, 1, 2],
...                    'B' : ['foo', 'bar', 'foo'],
```

(continues on next page)

(continued from previous page)

```

'C' : np.arange(4.,7)})

>>> df
   A   B   C
0  1  foo  4.0
1  1  bar  5.0
2  2  foo  6.0

```

```

>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (index: 3)
Coordinates:
  * index      (index) int64 0 1 2
Data variables:
  A            (index) int64 1 1 2
  B            (index) object 'foo' 'bar' 'foo'
  C            (index) float64 4.0 5.0 6.0

```

```

>>> df = pd.DataFrame({'A' : [1, 1, 2],
                        'B' : ['foo', 'bar', 'foo'],
                        'C' : np.arange(4.,7)})
                        ).set_index(['B', 'A'])

>>> df
      C
B   A
foo 1  4.0
bar 1  5.0
foo 2  6.0

```

```

>>> df.to_xarray()
<xarray.Dataset>
Dimensions:  (A: 2, B: 2)
Coordinates:
  * B          (B) object 'bar' 'foo'
  * A          (A) int64 1 2
Data variables:
  C            (B, A) float64 5.0 nan 4.0 6.0

```

```

>>> p = pd.Panel(np.arange(24).reshape(4,3,2),
                  items=list('ABCD'),
                  major_axis=pd.date_range('20130101', periods=3),
                  minor_axis=['first', 'second'])

>>> p
<class 'pandas.core.panel.Panel'>
Dimensions: 4 (items) x 3 (major_axis) x 2 (minor_axis)
Items axis: A to D
Major_axis axis: 2013-01-01 00:00:00 to 2013-01-03 00:00:00
Minor_axis axis: first to second

```

```

>>> p.to_xarray()
<xarray.DataArray (items: 4, major_axis: 3, minor_axis: 2)>
array([[[ 0,  1],
        [ 2,  3],
        [ 4,  5]],
       [[ 6,  7],
        [ 8,  9]],

```

(continues on next page)

(continued from previous page)

```

        [10, 11]],
        [[12, 13],
         [14, 15],
         [16, 17]]],
        [[18, 19],
         [20, 21],
         [22, 23]]])
Coordinates:
* items          (items) object 'A' 'B' 'C' 'D'
* major_axis     (major_axis) datetime64[ns] 2013-01-01 2013-01-02 2013-01-03_
→ # noqa
* minor_axis     (minor_axis) object 'first' 'second'

```

## pandas.Panel.transpose

Panel.**transpose** (\*args, \*\*kwargs)

Permute the dimensions of the Panel

### Parameters

**args** [three positional arguments: each one of]

**{0, 1, 2, 'items', 'major\_axis', 'minor\_axis'}**

**copy** [boolean, default False] Make a copy of the underlying data. Mixed-dtype data will always result in a copy

### Returns

**y** [same as input]

## Examples

```

>>> p.transpose(2, 0, 1)
>>> p.transpose(2, 0, 1, copy=True)

```

## pandas.Panel.truediv

Panel.**truediv** (other, axis=0)

Floating division of series and other, element-wise (binary operator *truediv*). Equivalent to `panel / other`.

### Parameters

**other** [DataFrame or Panel]

**axis** : {items, major\_axis, minor\_axis}

Axis to broadcast over

### Returns

**Panel**

See also:

[`Panel.rtruediv`](#)

## **pandas.Panel.truncate**

`Panel.truncate` (*before=None, after=None, axis=None, copy=True*)

Truncate a Series or DataFrame before and after some index value.

This is a useful shorthand for boolean indexing based on index values above or below certain thresholds.

**Parameters** **before** : date, string, int

Truncate all rows before this index value.

**after** : date, string, int

Truncate all rows after this index value.

**axis** : {0 or 'index', 1 or 'columns'}, optional

Axis to truncate. Truncates the index (rows) by default.

**copy** : boolean, default is True,

Return a copy of the truncated section.

**Returns** **type of caller**

The truncated Series or DataFrame.

See also:

[`DataFrame.loc`](#) Select a subset of a DataFrame by label.

[`DataFrame.iloc`](#) Select a subset of a DataFrame by position.

## **Notes**

If the index being truncated contains only datetime values, *before* and *after* may be specified as strings instead of Timestamps.

## **Examples**

```
>>> df = pd.DataFrame({'A': ['a', 'b', 'c', 'd', 'e'],
...                    'B': ['f', 'g', 'h', 'i', 'j'],
...                    'C': ['k', 'l', 'm', 'n', 'o']},
...                    index=[1, 2, 3, 4, 5])
>>> df
   A  B  C
1  a  f  k
2  b  g  l
3  c  h  m
4  d  i  n
5  e  j  o
```

```
>>> df.truncate(before=2, after=4)
   A  B  C
2  b  g  l
3  c  h  m
4  d  i  n
```

The columns of a DataFrame can be truncated.

```
>>> df.truncate(before="A", after="B", axis="columns")
   A  B
1  a  f
2  b  g
3  c  h
4  d  i
5  e  j
```

For Series, only rows can be truncated.

```
>>> df['A'].truncate(before=2, after=4)
2    b
3    c
4    d
Name: A, dtype: object
```

The index values in `truncate` can be datetimes or string dates.

```
>>> dates = pd.date_range('2016-01-01', '2016-02-01', freq='s')
>>> df = pd.DataFrame(index=dates, data={'A': 1})
>>> df.tail()
                A
2016-01-31 23:59:56  1
2016-01-31 23:59:57  1
2016-01-31 23:59:58  1
2016-01-31 23:59:59  1
2016-02-01 00:00:00  1
```

```
>>> df.truncate(before=pd.Timestamp('2016-01-05'),
...             after=pd.Timestamp('2016-01-10')).tail()
                A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Because the index is a `DatetimeIndex` containing only dates, we can specify *before* and *after* as strings. They will be coerced to `Timestamps` before truncation.

```
>>> df.truncate('2016-01-05', '2016-01-10').tail()
                A
2016-01-09 23:59:56  1
2016-01-09 23:59:57  1
2016-01-09 23:59:58  1
2016-01-09 23:59:59  1
2016-01-10 00:00:00  1
```

Note that `truncate` assumes a 0 value for any unspecified time component (midnight). This differs from partial string slicing, which returns any partially matching dates.



```
>>> df.loc['2016-01-05':'2016-01-10', :].tail()
      A
2016-01-10 23:59:55    1
2016-01-10 23:59:56    1
2016-01-10 23:59:57    1
2016-01-10 23:59:58    1
2016-01-10 23:59:59    1
```

**pandas.Panel.tshift**

`Panel.tshift` (*periods=1, freq=None, axis='major'*)

Shift the time index, using the index's frequency if available.

**Parameters** `periods` : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, default None

Increment to use from the tseries module or time rule (e.g. 'EOM')

**axis** : int or basestring

Corresponds to the axis that contains the Index

**Returns**

**shifted** [NDFrame]

**Notes**

If freq is not specified then tries to use the freq or inferred\_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

**pandas.Panel.tz\_convert**

`Panel.tz_convert` (*tz, axis=0, level=None, copy=True*)

Convert tz-aware axis to target time zone.

**Parameters**

**tz** [string or pytz.timezone object]

**axis** [the axis to convert]

**level** : int, str, default None

If axis is a MultiIndex, convert a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**Raises** **TypeError**

If the axis is tz-naive.

## pandas.Panel.tz\_localize

`Panel.tz_localize` (*tz*, *axis=0*, *level=None*, *copy=True*, *ambiguous='raise'*)

Localize tz-naive TimeSeries to target time zone.

### Parameters

**tz** [string or pytz.timezone object]

**axis** [the axis to localize]

**level** : int, str, default None

If axis is a MultiIndex, localize a specific level. Otherwise must be None

**copy** : boolean, default True

Also make a copy of the underlying data

**ambiguous** : 'infer', bool-ndarray, 'NaT', default 'raise'

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False designates a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

### Raises TypeError

If the TimeSeries is tz-aware and tz is not None.

## pandas.Panel.update

`Panel.update` (*other*, *join='left'*, *overwrite=True*, *filter\_func=None*, *raise\_conflict=False*)

Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel. Aligns on items

### Parameters

**other** [Panel, or object coercible to Panel]

**join** : How to join individual DataFrames

{ 'left', 'right', 'outer', 'inner' }, default 'left'

**overwrite** : boolean, default True

If True then overwrite values for common keys in the calling panel

**filter\_func** : callable(1d-array) -> 1d-array<boolean>, default None

Can choose to replace values other than NA. Return True for values that should be updated

**raise\_conflict** : bool

If True, will raise an error if a DataFrame and other both contain data in the same place.

**pandas.Panel.var**

`Panel.var` (*axis=None, skipna=None, level=None, ddof=1, numeric\_only=None, \*\*kwargs*)

Return unbiased variance over requested axis.

Normalized by N-1 by default. This can be changed using the `ddof` argument

**Parameters**

**axis** [{items (0), major\_axis (1), minor\_axis (2)}]

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**var** [DataFrame or Panel (if level specified)]

**pandas.Panel.where**

`Panel.where` (*cond, other=nan, inplace=False, axis=None, level=None, errors='raise', try\_cast=False, raise\_on\_error=None*)

Return an object of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

**Parameters** **cond** : boolean NDFrame, array-like, or callable

Where *cond* is True, keep the original value. Where False, replace with corresponding value from *other*. If *cond* is callable, it is computed on the NDFrame and should return boolean NDFrame or array. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as *cond*.

**other** : scalar, NDFrame, or callable

Entries where *cond* is False are replaced with corresponding value from *other*. If *other* is callable, it is computed on the NDFrame and should return scalar or NDFrame. The callable must not change input NDFrame (though pandas doesn't check it).

New in version 0.18.1: A callable can be used as *other*.

**inplace** : boolean, default False

Whether to perform the operation in place on the data

**axis** [alignment axis if needed, default None]

**level** [alignment level if needed, default None]

**errors** : str, {'raise', 'ignore'}, default 'raise'

- **raise** : allow exceptions to be raised
- **ignore** : suppress exceptions. On error return original object

Note that currently this parameter won't affect the results and will always coerce to a suitable dtype.

**try\_cast** : boolean, default False

try to cast the result back to the input type (if possible),

**raise\_on\_error** : boolean, default True

Whether to raise on invalid data types (e.g. trying to where on strings)

Deprecated since version 0.21.0.

### Returns

**wh** [same type as caller]

### See also:

[`DataFrame.mask\(\)`](#)

### Notes

The where method is an application of the if-then idiom. For each element in the calling DataFrame, if `cond` is `True` the element is used; otherwise the corresponding element from the DataFrame `other` is used.

The signature for [`DataFrame.where\(\)`](#) differs from [`numpy.where\(\)`](#). Roughly `df1.where(m, df2)` is equivalent to `np.where(m, df1, df2)`.

For further details and examples see the `where` documentation in [indexing](#).

### Examples

```
>>> s = pd.Series(range(5))
>>> s.where(s > 0)
0    NaN
1    1.0
2    2.0
3    3.0
4    4.0
```

```
>>> s.mask(s > 0)
0    0.0
1    NaN
2    NaN
3    NaN
4    NaN
```

```
>>> s.where(s > 1, 10)
0    10.0
1    10.0
2     2.0
3     3.0
4     4.0
```

```
>>> df = pd.DataFrame(np.arange(10).reshape(-1, 2), columns=['A', 'B'])
>>> m = df % 3 == 0
>>> df.where(m, -df)
   A  B
0  0 -1
1 -2  3
2 -4 -5
3  6 -7
4 -8  9
>>> df.where(m, -df) == np.where(m, df, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
>>> df.where(m, -df) == df.mask(~m, -df)
   A      B
0  True  True
1  True  True
2  True  True
3  True  True
4  True  True
```

## pandas.Panel.xs

`Panel.xs` (*key*, *axis=1*)

Return slice of panel along selected axis

**Parameters** *key*: object

Label

**axis** [{‘items’, ‘major’, ‘minor’}, default 1/‘major’]

**Returns**

*y* [ndim(self)-1]

## Notes

`xs` is only for getting, not setting values.

`MultiIndex Slicers` is a generic way to get/set values on any level or levels and is a superset of `xs` functionality, see [MultiIndex Slicers](#)

<b>agg</b>	
<b>aggregate</b>	
<b>drop</b>	

### 34.5.2 Attributes and underlying data

#### Axes

- **items**: axis 0; each item corresponds to a DataFrame contained inside
- **major\_axis**: axis 1; the index (rows) of each of the DataFrames
- **minor\_axis**: axis 2; the columns of each of the DataFrames

<i>Panel.values</i>	Return a Numpy representation of the DataFrame.
<i>Panel.axes</i>	Return index label(s) of the internal NDFrame
<i>Panel.ndim</i>	Return an int representing the number of axes / array dimensions.
<i>Panel.size</i>	Return an int representing the number of elements in this object.
<i>Panel.shape</i>	Return a tuple of axis dimensions
<i>Panel.dtypes</i>	Return the dtypes in the DataFrame.
<i>Panel.ftypes</i>	Return the ftypes (indication of sparse/dense and dtype) in DataFrame.
<i>Panel.get_dtype_counts()</i>	Return counts of unique dtypes in this object.
<i>Panel.get_ftype_counts()</i>	(DEPRECATED) Return counts of unique ftypes in this object.

### 34.5.3 Conversion

<i>Panel.astype(dtype[, copy, errors])</i>	Cast a pandas object to a specified dtype dtype.
<i>Panel.copy([deep])</i>	Make a copy of this object's indices and data.
<i>Panel.isna()</i>	Detect missing values.
<i>Panel.notna()</i>	Detect existing (non-missing) values.

### 34.5.4 Getting and setting

<i>Panel.get_value(*args, **kwargs)</i>	(DEPRECATED) Quickly retrieve single value at (item, major, minor) location
<i>Panel.set_value(*args, **kwargs)</i>	(DEPRECATED) Quickly set single value at (item, major, minor) location

### 34.5.5 Indexing, iteration, slicing

<i>Panel.at</i>	Access a single value for a row/column label pair.
<i>Panel.iat</i>	Access a single value for a row/column pair by integer position.

Continued on next page

Table 85 – continued from previous page

<i>Panel.loc</i>	Access a group of rows and columns by label(s) or a boolean array.
<i>Panel.iloc</i>	Purely integer-location based indexing for selection by position.
<i>Panel.__iter__()</i>	Iterate over infor axis
<i>Panel.iteritems()</i>	Iterate over (label, values) on info axis
<i>Panel.pop(item)</i>	Return item and drop from frame.
<i>Panel.xs(key[, axis])</i>	Return slice of panel along selected axis
<i>Panel.major_xs(key)</i>	Return slice of panel along major axis
<i>Panel.minor_xs(key)</i>	Return slice of panel along minor axis

#### 34.5.5.1 pandas.Panel.\_\_iter\_\_

`Panel.__iter__()`

Iterate over infor axis

For more information on `.at`, `.iat`, `.loc`, and `.iloc`, see the [indexing documentation](#).

### 34.5.6 Binary operator functions

<i>Panel.add(other[, axis])</i>	Addition of series and other, element-wise (binary operator <i>add</i> ).
<i>Panel.sub(other[, axis])</i>	Subtraction of series and other, element-wise (binary operator <i>sub</i> ).
<i>Panel.mul(other[, axis])</i>	Multiplication of series and other, element-wise (binary operator <i>mul</i> ).
<i>Panel.div(other[, axis])</i>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<i>Panel.truediv(other[, axis])</i>	Floating division of series and other, element-wise (binary operator <i>truediv</i> ).
<i>Panel.floordiv(other[, axis])</i>	Integer division of series and other, element-wise (binary operator <i>floordiv</i> ).
<i>Panel.mod(other[, axis])</i>	Modulo of series and other, element-wise (binary operator <i>mod</i> ).
<i>Panel.pow(other[, axis])</i>	Exponential power of series and other, element-wise (binary operator <i>pow</i> ).
<i>Panel.radd(other[, axis])</i>	Addition of series and other, element-wise (binary operator <i>radd</i> ).
<i>Panel.rsub(other[, axis])</i>	Subtraction of series and other, element-wise (binary operator <i>rsub</i> ).
<i>Panel.rmul(other[, axis])</i>	Multiplication of series and other, element-wise (binary operator <i>rmul</i> ).
<i>Panel.rdiv(other[, axis])</i>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<i>Panel.rtruediv(other[, axis])</i>	Floating division of series and other, element-wise (binary operator <i>rtruediv</i> ).
<i>Panel.rfloordiv(other[, axis])</i>	Integer division of series and other, element-wise (binary operator <i>rfloordiv</i> ).
<i>Panel.rmod(other[, axis])</i>	Modulo of series and other, element-wise (binary operator <i>rmod</i> ).

Continued on next page

Table 86 – continued from previous page

<code>Panel.rpow(other[, axis])</code>	Exponential power of series and other, element-wise (binary operator <i>rpow</i> ).
<code>Panel.lt(other[, axis])</code>	Wrapper for comparison method <code>lt</code>
<code>Panel.gt(other[, axis])</code>	Wrapper for comparison method <code>gt</code>
<code>Panel.le(other[, axis])</code>	Wrapper for comparison method <code>le</code>
<code>Panel.ge(other[, axis])</code>	Wrapper for comparison method <code>ge</code>
<code>Panel.ne(other[, axis])</code>	Wrapper for comparison method <code>ne</code>
<code>Panel.eq(other[, axis])</code>	Wrapper for comparison method <code>eq</code>

### 34.5.7 Function application, GroupBy

<code>Panel.apply(func[, axis])</code>	Applies function along axis (or axes) of the Panel
<code>Panel.groupby(function[, axis])</code>	Group data on given axis, returning GroupBy object

### 34.5.8 Computations / Descriptive Stats

<code>Panel.abs()</code>	Return a Series/DataFrame with absolute numeric value of each element.
<code>Panel.clip([lower, upper, axis, inplace])</code>	Trim values at input threshold(s).
<code>Panel.clip_lower(threshold[, axis, inplace])</code>	Return copy of the input with values below a threshold truncated.
<code>Panel.clip_upper(threshold[, axis, inplace])</code>	Return copy of input with values above given value(s) truncated.
<code>Panel.count([axis])</code>	Return number of observations over requested axis.
<code>Panel.cummax([axis, skipna])</code>	Return cumulative maximum over a DataFrame or Series axis.
<code>Panel.cummin([axis, skipna])</code>	Return cumulative minimum over a DataFrame or Series axis.
<code>Panel.cumprod([axis, skipna])</code>	Return cumulative product over a DataFrame or Series axis.
<code>Panel.cumsum([axis, skipna])</code>	Return cumulative sum over a DataFrame or Series axis.
<code>Panel.max([axis, skipna, level, numeric_only])</code>	This method returns the maximum of the values in the object.
<code>Panel.mean([axis, skipna, level, numeric_only])</code>	Return the mean of the values for the requested axis
<code>Panel.median([axis, skipna, level, numeric_only])</code>	Return the median of the values for the requested axis
<code>Panel.min([axis, skipna, level, numeric_only])</code>	This method returns the minimum of the values in the object.
<code>Panel.pct_change([periods, fill_method, ...])</code>	Percentage change between the current and a prior element.
<code>Panel.prod([axis, skipna, level, ...])</code>	Return the product of the values for the requested axis
<code>Panel.sem([axis, skipna, level, ddof, ...])</code>	Return unbiased standard error of the mean over requested axis.
<code>Panel.skew([axis, skipna, level, numeric_only])</code>	Return unbiased skew over requested axis Normalized by N-1
<code>Panel.sum([axis, skipna, level, ...])</code>	Return the sum of the values for the requested axis
<code>Panel.std([axis, skipna, level, ddof, ...])</code>	Return sample standard deviation over requested axis.
<code>Panel.var([axis, skipna, level, ddof, ...])</code>	Return unbiased variance over requested axis.



### 34.5.9 Reindexing / Selection / Label manipulation

<code>Panel.add_prefix(prefix)</code>	Prefix labels with string <i>prefix</i> .
<code>Panel.add_suffix(suffix)</code>	Suffix labels with string <i>suffix</i> .
<code>Panel.drop([labels, axis, index, columns, ...])</code>	
<code>Panel.equals(other)</code>	Determines if two NDFrame objects contain the same elements.
<code>Panel.filter([items, like, regex, axis])</code>	Subset rows or columns of dataframe according to labels in the specified index.
<code>Panel.first(offset)</code>	Convenience method for subsetting initial periods of time series data based on a date offset.
<code>Panel.last(offset)</code>	Convenience method for subsetting final periods of time series data based on a date offset.
<code>Panel.reindex(*args, **kwargs)</code>	Conform Panel to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>Panel.reindex_axis(labels[, axis, method, ...])</code>	Conform input object to new index with optional filling logic, placing NA/NaN in locations having no value in the previous index.
<code>Panel.reindex_like(other[, method, copy, ...])</code>	Return an object with matching indices to myself.
<code>Panel.rename([items, major_axis, minor_axis])</code>	Alter axes input function or functions.
<code>Panel.sample([n, frac, replace, weights, ...])</code>	Return a random sample of items from an axis of object.
<code>Panel.select(crit[, axis])</code>	(DEPRECATED) Return data corresponding to axis labels matching criteria
<code>Panel.take(indices[, axis, convert, is_copy])</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>Panel.truncate([before, after, axis, copy])</code>	Truncate a Series or DataFrame before and after some index value.

#### 34.5.9.1 pandas.Panel.drop

`Panel.drop(labels=None, axis=0, index=None, columns=None, level=None, inplace=False, errors='raise')`

### 34.5.10 Missing data handling

<code>Panel.dropna([axis, how, inplace])</code>	Drop 2D from panel, holding passed axis constant
---	--

### 34.5.11 Reshaping, sorting, transposing

<code>Panel.sort_index([axis, level, ascending, ...])</code>	Sort object by labels (along an axis)
<code>Panel.swaplevel([i, j, axis])</code>	Swap levels i and j in a MultiIndex on a particular axis
<code>Panel.transpose(*args, **kwargs)</code>	Permute the dimensions of the Panel
<code>Panel.swapaxes(axis1, axis2[, copy])</code>	Interchange axes and swap values axes appropriately
<code>Panel.conform(frame[, axis])</code>	Conform input DataFrame to align with chosen axis pair.

### 34.5.12 Combining / joining / merging

<code>Panel.join(other[, how, lsuffix, rsuffix])</code>	Join items with other Panel either on major and minor axes column
<code>Panel.update(other[, join, overwrite, ...])</code>	Modify Panel in place using non-NA values from passed Panel, or object coercible to Panel.

### 34.5.13 Time series-related

<code>Panel.asfreq(freq[, method, how, normalize, ...])</code>	Convert TimeSeries to specified frequency.
<code>Panel.shift([periods, freq, axis])</code>	Shift index by desired number of periods with an optional time freq.
<code>Panel.resample(rule[, how, axis, ...])</code>	Convenience method for frequency conversion and re-sampling of time series.
<code>Panel.tz_convert(tz[, axis, level, copy])</code>	Convert tz-aware axis to target time zone.
<code>Panel.tz_localize(tz[, axis, level, copy, ...])</code>	Localize tz-naive TimeSeries to target time zone.

### 34.5.14 Serialization / IO / Conversion

<code>Panel.from_dict(data[, intersect, orient, dtype])</code>	Construct Panel from dict of DataFrame objects
<code>Panel.to_pickle(path[, compression, protocol])</code>	Pickle (serialize) object to file.
<code>Panel.to_excel(path[, na_rep, engine])</code>	Write each DataFrame in Panel to a separate excel sheet
<code>Panel.to_hdf(path_or_buf, key, **kwargs)</code>	Write the contained data to an HDF5 file using HDFStore.
<code>Panel.to_sparse(*args, **kwargs)</code>	NOT IMPLEMENTED: do not call this method, as sparsifying is not supported for Panel objects and will raise an error.
<code>Panel.to_frame([filter_observations])</code>	Transform wide format into long (stacked) format as DataFrame whose columns are the Panel's items and whose index is a MultiIndex formed of the Panel's major and minor axes.
<code>Panel.to_clipboard([excel, sep])</code>	Copy object to the system clipboard.

## 34.6 Index

Many of these methods or variants thereof are available on the objects that contain an index (Series/DataFrame) and those should most likely be used before calling these methods directly.

<code>Index</code>	Immutable ndarray implementing an ordered, sliceable set.
--------------------	---

### 34.6.1 pandas.Index

**class** `pandas.Index`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects

**Parameters**

**data** [array-like (1-dimensional)]

**dtype** : NumPy dtype (default: object)

If dtype is None, we find the dtype that best fits the data. If an actual dtype is provided, we coerce to that dtype if it's safe. Otherwise, an error will be raised.

**copy** : bool

Make a copy of input ndarray

**name** : object

Name to be stored in the index

**tupleize\_cols** : bool (default: True)

When True, attempt to create a MultiIndex if possible

See also:

**RangeIndex** Index implementing a monotonic integer range

**CategoricalIndex** Index of *Categorical* s.

**MultiIndex** A multi-level, or hierarchical, Index

**IntervalIndex** an Index of *Interval* s.

*DatetimeIndex*, *TimedeltaIndex*, *PeriodIndex*, *Int64Index*, *UInt64Index*,  
*Float64Index*

## Notes

An Index instance can **only** contain hashable objects

## Examples

```
>>> pd.Index([1, 2, 3])
Int64Index([1, 2, 3], dtype='int64')
```

```
>>> pd.Index(list('abc'))
Index(['a', 'b', 'c'], dtype='object')
```

## Attributes

<i>T</i>	return the transpose, which is by definition self
<i>base</i>	return the base object if the memory of the underlying data is shared
<i>data</i>	return the data pointer of the underlying data
<i>dtype</i>	return the dtype object of the underlying data
<i>dtype_str</i>	return the dtype str of the underlying data
<i>flags</i>	

Continued on next page

Table 96 – continued from previous page

<i>hasnans</i>	return if I have any nans; enables various perf speedups
<i>inferred_type</i>	return a string of the type inferred from the values
<i>is_monotonic</i>	alias for <i>is_monotonic_increasing</i> (deprecated)
<i>is_monotonic_decreasing</i>	return if the index is monotonic decreasing (only equal or decreasing) values.
<i>is_monotonic_increasing</i>	return if the index is monotonic increasing (only equal or increasing) values.
<i>is_unique</i>	return if the index has unique values
<i>itemsize</i>	return the size of the dtype of the item of the underlying data
<i>nbytes</i>	return the number of bytes in the underlying data
<i>ndim</i>	return the number of dimensions of the underlying data, by definition 1
<i>shape</i>	return a tuple of the shape of the underlying data
<i>size</i>	return the number of elements in the underlying data
<i>strides</i>	return the strides of the underlying data
<i>values</i>	return the underlying data as an ndarray

#### 34.6.1.1 pandas.Index.T

`Index.T`  
return the transpose, which is by definition self

#### 34.6.1.2 pandas.Index.base

`Index.base`  
return the base object if the memory of the underlying data is shared

#### 34.6.1.3 pandas.Index.data

`Index.data`  
return the data pointer of the underlying data

#### 34.6.1.4 pandas.Index.dtype

`Index.dtype`  
return the dtype object of the underlying data

#### 34.6.1.5 pandas.Index.dtype\_str

`Index.dtype_str`  
return the dtype str of the underlying data

#### 34.6.1.6 pandas.Index.flags

`Index.flags`

### 34.6.1.7 pandas.Index.hasnans

`Index.hasnans`

return if I have any nans; enables various perf speedups

### 34.6.1.8 pandas.Index.inferred\_type

`Index.inferred_type`

return a string of the type inferred from the values

### 34.6.1.9 pandas.Index.is\_monotonic

`Index.is_monotonic`

alias for `is_monotonic_increasing` (deprecated)

### 34.6.1.10 pandas.Index.is\_monotonic\_decreasing

`Index.is_monotonic_decreasing`

return if the index is monotonic decreasing (only equal or decreasing) values.

#### Examples

```
>>> Index([3, 2, 1]).is_monotonic_decreasing
True
>>> Index([3, 2, 2]).is_monotonic_decreasing
True
>>> Index([3, 1, 2]).is_monotonic_decreasing
False
```

### 34.6.1.11 pandas.Index.is\_monotonic\_increasing

`Index.is_monotonic_increasing`

return if the index is monotonic increasing (only equal or increasing) values.

#### Examples

```
>>> Index([1, 2, 3]).is_monotonic_increasing
True
>>> Index([1, 2, 2]).is_monotonic_increasing
True
>>> Index([1, 3, 2]).is_monotonic_increasing
False
```

### 34.6.1.12 pandas.Index.is\_unique

`Index.is_unique`

return if the index has unique values

#### 34.6.1.13 pandas.Index.itemsize

`Index.itemsize`

return the size of the dtype of the item of the underlying data

#### 34.6.1.14 pandas.Index.nbytes

`Index.nbytes`

return the number of bytes in the underlying data

#### 34.6.1.15 pandas.Index.ndim

`Index.ndim`

return the number of dimensions of the underlying data, by definition 1

#### 34.6.1.16 pandas.Index.shape

`Index.shape`

return a tuple of the shape of the underlying data

#### 34.6.1.17 pandas.Index.size

`Index.size`

return the number of elements in the underlying data

#### 34.6.1.18 pandas.Index.strides

`Index.strides`

return the strides of the underlying data

#### 34.6.1.19 pandas.Index.values

`Index.values`

return the underlying data as an ndarray

<b>asi8</b>	
<b>empty</b>	
<b>has_duplicates</b>	
<b>is_all_dates</b>	
<b>name</b>	
<b>names</b>	
<b>nlevels</b>	

### Methods

---

`all(*args, **kwargs)`

Return whether all elements are True.

Continued on next page

Table 97 – continued from previous page

<code>any(*args, **kwargs)</code>	Return whether any element is True.
<code>append(other)</code>	Append a collection of Index options together
<code>argmax([axis])</code>	return a ndarray of the maximum argument indexer
<code>argmin([axis])</code>	return a ndarray of the minimum argument indexer
<code>argsort(*args, **kwargs)</code>	Return the integer indicies that would sort the index.
<code>asof(label)</code>	For a sorted index, return the most recent label up to and including the passed label.
<code>asof_locs(where, mask)</code>	where : array of timestamps mask : array of booleans where data is not NA
<code>astype(dtype[, copy])</code>	Create an Index with values cast to dtypes.
<code>contains(key)</code>	return a boolean if this key is IN the index
<code>copy([name, deep, dtype])</code>	Make a copy of this object.
<code>delete(loc)</code>	Make new Index with passed location(-s) deleted
<code>difference(other)</code>	Return a new Index with elements from the index that are not in <i>other</i> .
<code>drop(labels[, errors])</code>	Make new Index with passed list of labels deleted
<code>drop_duplicates([keep])</code>	Return Index with duplicate values removed.
<code>dropna([how])</code>	Return Index without NA/NaN values
<code>duplicated([keep])</code>	Indicate duplicate index values.
<code>equals(other)</code>	Determines if two Index objects contain the same elements.
<code>factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>fillna([value, downcast])</code>	Fill NA/NaN values with the specified value
<code>format([name, formatter])</code>	Render a string representation of the Index
<code>get_duplicates()</code>	(DEPRECATED) Extract duplicated index elements.
<code>get_indexer(target[, method, limit, tolerance])</code>	Compute indexer and mask for new index given the current index.
<code>get_indexer_for(target, **kwargs)</code>	guaranteed return of an indexer even when non-unique This dispatches to <code>get_indexer</code> or <code>get_indexer_nonunique</code> as appropriate
<code>get_indexer_non_unique(target)</code>	Compute indexer and mask for new index given the current index.
<code>get_level_values(level)</code>	Return an Index of values for requested level, equal to the length of the index.
<code>get_loc(key[, method, tolerance])</code>	Get integer location, slice or boolean mask for requested label.
<code>get_slice_bound(label, side, kind)</code>	Calculate slice bound that corresponds to given label.
<code>get_value(series, key)</code>	Fast lookup of value from 1-dimensional ndarray.
<code>get_values()</code>	Return <i>Index</i> data as an <i>numpy.ndarray</i> .
<code>groupby(values)</code>	Group the index labels by a given array of values.
<code>identical(other)</code>	Similar to <code>equals</code> , but check that other comparable attributes are also equal
<code>insert(loc, item)</code>	Make new Index inserting new item at location.
<code>intersection(other)</code>	Form the intersection of two Index objects.
<code>is_(other)</code>	More flexible, faster check like <code>is</code> but that works through views
<code>is_categorical()</code>	Check if the Index holds categorical data.
<code>isin(values[, level])</code>	Return a boolean array where the index values are in <i>values</i> .
<code>isna()</code>	Detect missing values.

Continued on next page

Table 97 – continued from previous page

<code>isnull()</code>	Detect missing values.
<code>item()</code>	return the first element of the underlying data as a python scalar
<code>join(other[, how, level, return_indexers, sort])</code>	<i>this is an internal non-public method</i>
<code>map(mapper[, na_action])</code>	Map values using input correspondence (a dict, Series, or function).
<code>max()</code>	Return the maximum value of the Index.
<code>memory_usage([deep])</code>	Memory usage of the values
<code>min()</code>	Return the minimum value of the Index.
<code>notna()</code>	Detect existing (non-missing) values.
<code>notnull()</code>	Detect existing (non-missing) values.
<code>nunique([dropna])</code>	Return number of unique elements in the object.
<code>putmask(mask, value)</code>	return a new Index of the values set with the mask
<code>ravel([order])</code>	return an ndarray of the flattened values of the underlying data
<code>reindex(target[, method, level, limit, ...])</code>	Create index with target's values (move/add/delete values as necessary)
<code>rename(name[, inplace])</code>	Set new names on index.
<code>repeat(repeats, *args, **kwargs)</code>	Repeat elements of an Index.
<code>searchsorted(value[, side, sorter])</code>	Find indices where elements should be inserted to maintain order.
<code>set_names(names[, level, inplace])</code>	Set new names on index.
<code>set_value(arr, key, value)</code>	Fast lookup of value from 1-dimensional ndarray.
<code>shift([periods, freq])</code>	Shift index by desired number of time frequency increments.
<code>slice_indexer([start, end, step, kind])</code>	For an ordered or unique index, compute the slice indexer for input labels and step.
<code>slice_locs([start, end, step, kind])</code>	Compute slice locations for input labels.
<code>sort_values([return_indexer, ascending])</code>	Return a sorted copy of the index.
<code>sortlevel([level, ascending, sort_remaining])</code>	For internal compatibility with with the Index API
<code>str</code>	alias of <code>pandas.core.strings.StringMethods</code>
<code>summary([name])</code>	(DEPRECATED) Return a summarized representation ..
<code>symmetric_difference(other[, result_name])</code>	Compute the symmetric difference of two Index objects.
<code>take(indices[, axis, allow_fill, fill_value])</code>	return a new Index of the values selected by the indices
<code>to_frame([index])</code>	Create a DataFrame with a column containing the Index.
<code>to_native_types([slicer])</code>	Format specified values of <i>self</i> and return them.
<code>to_series([index, name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
<code>tolist()</code>	Return a list of the values.
<code>transpose(*args, **kwargs)</code>	return the transpose, which is by definition self
<code>union(other)</code>	Form the union of two Index objects and sorts if possible.
<code>unique([level])</code>	Return unique values in the index.
<code>value_counts([normalize, sort, ascending, ...])</code>	Returns object containing counts of unique values.
<code>where(cond[, other])</code>	



### 34.6.1.20 pandas.Index.all

`Index.all(*args, **kwargs)`

Return whether all elements are True.

**Parameters** `*args`

These parameters will be passed to `numpy.all`.

**`**kwargs`**

These parameters will be passed to `numpy.all`.

**Returns** `all` : bool or array\_like (if axis is specified)

A single element array\_like may be converted to bool.

**See also:**

[`pandas.Index.any`](#) Return whether any element in an Index is True.

[`pandas.Series.any`](#) Return whether any element in a Series is True.

[`pandas.Series.all`](#) Return whether all elements in a Series are True.

#### Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

#### Examples

**all**

True, because nonzero integers are considered True.

```
>>> pd.Index([1, 2, 3]).all()
True
```

False, because 0 is considered False.

```
>>> pd.Index([0, 1, 2]).all()
False
```

**any**

True, because 1 is considered True.

```
>>> pd.Index([0, 0, 1]).any()
True
```

False, because 0 is considered False.

```
>>> pd.Index([0, 0, 0]).any()
False
```

### 34.6.1.21 pandas.Index.any

`Index.any(*args, **kwargs)`

Return whether any element is True.

#### Parameters *\*args*

These parameters will be passed to `numpy.any`.

#### *\*\*kwargs*

These parameters will be passed to `numpy.any`.

**Returns** `any` : bool or array\_like (if axis is specified)

A single element array\_like may be converted to bool.

**See also:**

[`pandas.Index.all`](#) Return whether all elements are True.

[`pandas.Series.all`](#) Return whether all elements are True.

#### Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

#### Examples

```
>>> index = pd.Index([0, 1, 2])
>>> index.any()
True
```

```
>>> index = pd.Index([0, 0, 0])
>>> index.any()
False
```

### 34.6.1.22 pandas.Index.append

`Index.append(other)`

Append a collection of Index options together

#### Parameters

**other** [Index or list/tuple of indices]

#### Returns

**appended** [Index]

### 34.6.1.23 pandas.Index.argmax

`Index.argmax(axis=None)`

return a ndarray of the maximum argument indexer

See also:

`numpy.ndarray.argmax`

#### 34.6.1.24 `pandas.Index.argmax`

`Index.argmax` (*axis=None*)

return a ndarray of the minimum argument indexer

See also:

`numpy.ndarray.argmin`

#### 34.6.1.25 `pandas.Index.argsort`

`Index.argsort` (*\*args, \*\*kwargs*)

Return the integer indicies that would sort the index.

**Parameters** *\*args*

Passed to `numpy.ndarray.argsort`.

***\*\*kwargs***

Passed to `numpy.ndarray.argsort`.

**Returns** `numpy.ndarray`

Integer indicies that would sort the index if used as an indexer.

See also:

`numpy.argsort` Similar method for NumPy arrays.

`Index.sort_values` Return sorted copy of Index.

#### Examples

```
>>> idx = pd.Index(['b', 'a', 'd', 'c'])
>>> idx
Index(['b', 'a', 'd', 'c'], dtype='object')
```

```
>>> order = idx.argsort()
>>> order
array([1, 0, 3, 2])
```

```
>>> idx[order]
Index(['a', 'b', 'c', 'd'], dtype='object')
```

#### 34.6.1.26 `pandas.Index.asof`

`Index.asof` (*label*)

For a sorted index, return the most recent label up to and including the passed label. Return NaN if not found.

See also:

`get_loc` asof is a thin wrapper around `get_loc` with `method='pad'`

#### 34.6.1.27 pandas.Index.asof\_locs

`Index.asof_locs` (*where, mask*)

*where* : array of timestamps *mask* : array of booleans where data is not NA

#### 34.6.1.28 pandas.Index.astype

`Index.astype` (*dtype, copy=True*)

Create an Index with values cast to dtypes. The class of a new Index is determined by dtype. When conversion is impossible, a `ValueError` exception is raised.

##### Parameters

**dtype** [numpy dtype or pandas type]

**copy** : bool, default True

By default, `astype` always returns a newly allocated object. If `copy` is set to `False` and internal requirements on dtype are satisfied, the original data is used to create a new Index or the original Index is returned.

New in version 0.19.0.

#### 34.6.1.29 pandas.Index.contains

`Index.contains` (*key*)

return a boolean if this key is IN the index

##### Parameters

**key** [object]

##### Returns

**boolean**

#### 34.6.1.30 pandas.Index.copy

`Index.copy` (*name=None, deep=False, dtype=None, \*\*kwargs*)

Make a copy of this object. Name and dtype sets those attributes on the new object.

##### Parameters

**name** [string, optional]

**deep** [boolean, default False]

**dtype** [numpy dtype or pandas type]

##### Returns

**copy** [Index]

## Notes

In most cases, there should be no functional difference from using `deep`, but if `deep` is passed it will attempt to deepcopy.

### 34.6.1.31 `pandas.Index.delete`

`Index.delete` (*loc*)

Make new Index with passed location(-s) deleted

#### Returns

**new\_index** [Index]

### 34.6.1.32 `pandas.Index.difference`

`Index.difference` (*other*)

Return a new Index with elements from the index that are not in *other*.

This is the set difference of two Index objects. It's sorted if sorting is possible.

#### Parameters

**other** [Index or array-like]

#### Returns

**difference** [Index]

## Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.difference(idx2)
Int64Index([1, 2], dtype='int64')
```

### 34.6.1.33 `pandas.Index.drop`

`Index.drop` (*labels*, *errors='raise'*)

Make new Index with passed list of labels deleted

#### Parameters

**labels** [array-like]

**errors**: {'ignore', 'raise'}, default 'raise'

If 'ignore', suppress error and existing labels are dropped.

#### Returns

**dropped** [Index]

#### Raises `KeyError`

If not all of the labels are found in the selected axis

### 34.6.1.34 pandas.Index.drop\_duplicates

`Index.drop_duplicates` (*keep*='first')

Return Index with duplicate values removed.

**Parameters** *keep* : {'first', 'last', False}, default 'first'

- 'first' : Drop duplicates except for the first occurrence.
- 'last' : Drop duplicates except for the last occurrence.
- False : Drop all duplicates.

**Returns**

**deduplicated** [Index]

See also:

*Series.drop\_duplicates* equivalent method on Series

*DataFrame.drop\_duplicates* equivalent method on DataFrame

*Index.duplicated* related method on Index, indicating duplicate Index values.

#### Examples

Generate an pandas.Index with duplicate values.

```
>>> idx = pd.Index(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'])
```

The *keep* parameter controls which duplicate values are removed. The value 'first' keeps the first occurrence for each set of duplicated entries. The default value of *keep* is 'first'.

```
>>> idx.drop_duplicates(keep='first')
Index(['lama', 'cow', 'beetle', 'hippo'], dtype='object')
```

The value 'last' keeps the last occurrence for each set of duplicated entries.

```
>>> idx.drop_duplicates(keep='last')
Index(['cow', 'beetle', 'lama', 'hippo'], dtype='object')
```

The value False discards all sets of duplicated entries.

```
>>> idx.drop_duplicates(keep=False)
Index(['cow', 'beetle', 'hippo'], dtype='object')
```

### 34.6.1.35 pandas.Index.dropna

`Index.dropna` (*how*='any')

Return Index without NA/NaN values

**Parameters** *how* : {'any', 'all'}, default 'any'

If the Index is a MultiIndex, drop the value when any or all levels are NaN.

**Returns**

**valid** [Index]

### 34.6.1.36 pandas.Index.duplicated

`Index.duplicated` (*keep*='first')

Indicate duplicate index values.

Duplicated values are indicated as `True` values in the resulting array. Either all duplicates, all except the first, or all except the last occurrence of duplicates can be indicated.

**Parameters** `keep` : { 'first', 'last', False }, default 'first'

The value or values in a set of duplicates to mark as missing.

- 'first' : Mark duplicates as `True` except for the first occurrence.
- 'last' : Mark duplicates as `True` except for the last occurrence.
- False : Mark all duplicates as `True`.

**Returns**

`numpy.ndarray`

See also:

[`pandas.Series.duplicated`](#) Equivalent method on `pandas.Series`

[`pandas.DataFrame.duplicated`](#) Equivalent method on `pandas.DataFrame`

[`pandas.Index.drop\_duplicates`](#) Remove duplicate values from Index

### Examples

By default, for each set of duplicated values, the first occurrence is set to `False` and all others to `True`:

```
>>> idx = pd.Index(['lama', 'cow', 'lama', 'beetle', 'lama'])
>>> idx.duplicated()
array([False, False,  True, False,  True])
```

which is equivalent to

```
>>> idx.duplicated(keep='first')
array([False, False,  True, False,  True])
```

By using 'last', the last occurrence of each set of duplicated values is set on `False` and all others on `True`:

```
>>> idx.duplicated(keep='last')
array([ True, False,  True, False, False])
```

By setting `keep` on `False`, all duplicates are `True`:

```
>>> idx.duplicated(keep=False)
array([ True, False,  True, False,  True])
```

### 34.6.1.37 pandas.Index.equals

`Index.equals` (*other*)

Determines if two Index objects contain the same elements.

### 34.6.1.38 pandas.Index.factorize

`Index.factorize` (*sort=False*, *na\_sentinel=-1*)

Encode the object as an enumerated type or categorical variable.

This method is useful for obtaining a numeric representation of an array when all that matters is identifying distinct values. *factorize* is available as both a top-level function `pandas.factorize()`, and as a method `Series.factorize()` and `Index.factorize()`.

**Parameters** *sort* : boolean, default False

Sort *uniques* and shuffle *labels* to maintain the relationship.

*na\_sentinel* : int, default -1

Value to mark “not found”.

**Returns** *labels* : ndarray

An integer ndarray that’s an indexer into *uniques*. `uniques.take(labels)` will have the same values as *values*.

*uniques* : ndarray, Index, or Categorical

The unique valid values. When *values* is Categorical, *uniques* is a Categorical. When *values* is some other pandas object, an *Index* is returned. Otherwise, a 1-D ndarray is returned.

---

**Note:** Even if there’s a missing value in *values*, *uniques* will *not* contain an entry for it.

---

See also:

`pandas.cut` Discretize continuous-valued array.

`pandas.unique` Find the unique value in an array.

### Examples

These examples all show *factorize* as a top-level method like `pd.factorize(values)`. The results are identical for methods like `Series.factorize()`.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'])
>>> labels
array([0, 0, 1, 2, 0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

With *sort=True*, the *uniques* will be sorted, and *labels* will be shuffled so that the relationship is the maintained.

```
>>> labels, uniques = pd.factorize(['b', 'b', 'a', 'c', 'b'], sort=True)
>>> labels
array([1, 1, 0, 2, 1])
>>> uniques
array(['a', 'b', 'c'], dtype=object)
```



Missing values are indicated in *labels* with *na\_sentinel* (-1 by default). Note that missing values are never included in *uniques*.

```
>>> labels, uniques = pd.factorize(['b', None, 'a', 'c', 'b'])
>>> labels
array([ 0, -1,  1,  2,  0])
>>> uniques
array(['b', 'a', 'c'], dtype=object)
```

Thus far, we've only factorized lists (which are internally coerced to NumPy arrays). When factorizing pandas objects, the type of *uniques* will differ. For Categoricals, a *Categorical* is returned.

```
>>> cat = pd.Categorical(['a', 'a', 'c'], categories=['a', 'b', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
[a, c]
Categories (3, object): [a, b, c]
```

Notice that 'b' is in `uniques.categories`, despite not being present in `cat.values`.

For all other pandas objects, an Index of the appropriate type is returned.

```
>>> cat = pd.Series(['a', 'a', 'c'])
>>> labels, uniques = pd.factorize(cat)
>>> labels
array([0, 0, 1])
>>> uniques
Index(['a', 'c'], dtype='object')
```

### 34.6.1.39 pandas.Index.fillna

`Index.fillna` (*value=None, downcast=None*)

Fill NA/NaN values with the specified value

**Parameters** *value* : scalar

Scalar value to use to fill holes (e.g. 0). This value cannot be a list-likes.

**downcast** : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

**Returns**

**filled** [%(*klass*)s]

### 34.6.1.40 pandas.Index.format

`Index.format` (*name=False, formatter=None, \*\*kwargs*)

Render a string representation of the Index

### 34.6.1.41 pandas.Index.get\_duplicates

`Index.get_duplicates()`

Extract duplicated index elements.

Returns a sorted list of index elements which appear more than once in the index.

Deprecated since version 0.23.0: Use `idx[idx.duplicated()].unique()` instead

**Returns array-like**

List of duplicated indexes.

**See also:**

[`Index.duplicated`](#) Return boolean array denoting duplicates.

[`Index.drop\_duplicates`](#) Return Index with duplicates removed.

### Examples

Works on different Index of types.

```
>>> pd.Index([1, 2, 2, 3, 3, 3, 4]).get_duplicates()
[2, 3]
>>> pd.Index([1., 2., 2., 3., 3., 3., 4.]).get_duplicates()
[2.0, 3.0]
>>> pd.Index(['a', 'b', 'b', 'c', 'c', 'c', 'd']).get_duplicates()
['b', 'c']
```

Note that for a `DatetimeIndex`, it does not return a list but a new `DatetimeIndex`:

```
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-02', '2018-01-03',
...                          '2018-01-03', '2018-01-04', '2018-01-04'],
...                          format='%Y-%m-%d')
>>> pd.Index(dates).get_duplicates()
DatetimeIndex(['2018-01-03', '2018-01-04'],
              dtype='datetime64[ns]', freq=None)
```

Sorts duplicated elements even when indexes are unordered.

```
>>> pd.Index([1, 2, 3, 2, 3, 4, 3]).get_duplicates()
[2, 3]
```

Return empty array-like structure when all elements are unique.

```
>>> pd.Index([1, 2, 3, 4]).get_duplicates()
[]
>>> dates = pd.to_datetime(['2018-01-01', '2018-01-02', '2018-01-03'],
...                          format='%Y-%m-%d')
>>> pd.Index(dates).get_duplicates()
DatetimeIndex([], dtype='datetime64[ns]', freq=None)
```

### 34.6.1.42 pandas.Index.get\_indexer

`Index.get_indexer(target, method=None, limit=None, tolerance=None)`

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to `ndarray.take` to align the current data to the new index.

**Parameters****target** [Index]**method** : {None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

**limit** : int, optional

Maximum number of consecutive labels in `target` to match for inexact matches.

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation  $\text{abs}(\text{index}[\text{indexer}] - \text{target}) \leq \text{tolerance}$ .

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

**Returns** **indexer** : ndarray of int

Integers from 0 to `n - 1` indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

**Examples**

```
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

**34.6.1.43 pandas.Index.get\_indexer\_for**`Index.get_indexer_for(target, **kwargs)`

guaranteed return of an indexer even when non-unique This dispatches to `get_indexer` or `get_indexer_nonunique` as appropriate

**34.6.1.44 pandas.Index.get\_indexer\_non\_unique**`Index.get_indexer_non_unique(target)`

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to `ndarray.take` to align the current data to the new index.

**Parameters****target** [Index]

**Returns indexer** : ndarray of int

Integers from 0 to n - 1 indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

**missing** : ndarray of int

An indexer into the target of the values not found. These correspond to the -1 in the indexer array

#### 34.6.1.45 pandas.Index.get\_level\_values

`Index.get_level_values (level)`

Return an Index of values for requested level, equal to the length of the index.

**Parameters level** : int or str

level is either the integer position of the level in the MultiIndex, or the name of the level.

**Returns values** : Index

self, as there is only one level in the Index.

**See also:**

[`pandas.MultiIndex.get\_level\_values`](#) get values for a level of a MultiIndex

#### 34.6.1.46 pandas.Index.get\_loc

`Index.get_loc (key, method=None, tolerance=None)`

Get integer location, slice or boolean mask for requested label.

**Parameters**

**key** [label]

**method** : {None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest' }, optional

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

**tolerance** : optional

Maximum distance from index value for inexact matches. The value of the index at the matching location must satisfy the equation `abs(index[loc] - key) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

**Returns**

**loc** [int if unique index, slice if monotonic index, else mask]

## Examples

```
>>> unique_index = pd.Index(list('abc'))
>>> unique_index.get_loc('b')
1
```

```
>>> monotonic_index = pd.Index(list('abbc'))
>>> monotonic_index.get_loc('b')
slice(1, 3, None)
```

```
>>> non_monotonic_index = pd.Index(list('abcb'))
>>> non_monotonic_index.get_loc('b')
array([False,  True, False,  True], dtype=bool)
```

### 34.6.1.47 pandas.Index.get\_slice\_bound

`Index.get_slice_bound(label, side, kind)`

Calculate slice bound that corresponds to given label.

Returns leftmost (one-past-the-rightmost if `side=='right'`) position of given label.

#### Parameters

**label** [object]

**side** [{ 'left', 'right' }]

**kind** [{ 'ix', 'loc', 'getitem' }]

### 34.6.1.48 pandas.Index.get\_value

`Index.get_value(series, key)`

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

### 34.6.1.49 pandas.Index.get\_values

`Index.get_values()`

Return *Index* data as an *numpy.ndarray*.

#### Returns numpy.ndarray

A one-dimensional numpy array of the *Index* values.

#### See also:

[\*Index.values\*](#) The attribute that `get_values` wraps.

## Examples

Getting the *Index* values of a *DataFrame*:

```
>>> df = pd.DataFrame([[1, 2, 3], [4, 5, 6], [7, 8, 9]],
...                    index=['a', 'b', 'c'], columns=['A', 'B', 'C'])
>>> df
   A  B  C
a  1  2  3
b  4  5  6
c  7  8  9
>>> df.index.get_values()
array(['a', 'b', 'c'], dtype=object)
```

Standalone *Index* values:

```
>>> idx = pd.Index(['1', '2', '3'])
>>> idx.get_values()
array(['1', '2', '3'], dtype=object)
```

*MultiIndex* arrays also have only one dimension:

```
>>> midx = pd.MultiIndex.from_arrays([[1, 2, 3], ['a', 'b', 'c']],
...                                 names=('number', 'letter'))
>>> midx.get_values()
array([(1, 'a'), (2, 'b'), (3, 'c')], dtype=object)
>>> midx.get_values().ndim
1
```

#### 34.6.1.50 pandas.Index.groupby

`Index.groupby` (*values*)

Group the index labels by a given array of values.

**Parameters** `values`: array

Values used to determine the groups.

**Returns** `groups`: dict

{group name -> group labels}

#### 34.6.1.51 pandas.Index.identical

`Index.identical` (*other*)

Similar to equals, but check that other comparable attributes are also equal

#### 34.6.1.52 pandas.Index.insert

`Index.insert` (*loc*, *item*)

Make new Index inserting new item at location. Follows Python list.append semantics for negative values

**Parameters**

`loc` [int]

`item` [object]

**Returns**

`new_index` [Index]

### 34.6.1.53 pandas.Index.intersection

`Index.intersection(other)`

Form the intersection of two Index objects.

This returns a new Index with elements common to the index and *other*, preserving the order of the calling index.

**Parameters**

**other** [Index or array-like]

**Returns**

**intersection** [Index]

#### Examples

```

>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.intersection(idx2)
Int64Index([3, 4], dtype='int64')

```

### 34.6.1.54 pandas.Index.is\_

`Index.is_(other)`

More flexible, faster check like `is` but that works through views

Note: this is *not* the same as `Index.identical()`, which checks that metadata is also the same.

**Parameters** **other** : object

other object to compare against.

**Returns**

**True if both have same underlying data, False otherwise** [bool]

### 34.6.1.55 pandas.Index.is\_categorical

`Index.is_categorical()`

Check if the Index holds categorical data.

**Returns** **boolean**

True if the Index is categorical.

**See also:**

[\*CategoricalIndex\*](#) Index for categorical data.

#### Examples

```
>>> idx = pd.Index(["Watermelon", "Orange", "Apple",
...                 "Watermelon"]).astype("category")
>>> idx.is_categorical()
True
```

```
>>> idx = pd.Index([1, 3, 5, 7])
>>> idx.is_categorical()
False
```

```
>>> s = pd.Series(["Peter", "Victor", "Elisabeth", "Mar"])
>>> s
0      Peter
1    Victor
2  Elisabeth
3        Mar
dtype: object
>>> s.index.is_categorical()
False
```

#### 34.6.1.56 pandas.Index.isin

`Index.isin` (*values*, *level=None*)

Return a boolean array where the index values are in *values*.

Compute boolean array of whether each index value is found in the passed set of values. The length of the returned boolean array matches the length of the index.

**Parameters** *values* : set or list-like

Sought values.

New in version 0.18.1: Support for values as a set.

**level** : str or int, optional

Name or position of the index level to use (if the index is a *MultiIndex*).

**Returns** *is\_contained* : ndarray

NumPy array of boolean values.

**See also:**

[\*Series.isin\*](#) Same for Series.

[\*DataFrame.isin\*](#) Same method for DataFrames.

#### Notes

In the case of *MultiIndex* you must either specify *values* as a list-like object containing tuples that are the same length as the number of levels, or specify *level*. Otherwise it will raise a `ValueError`.

If *level* is specified:

- if it is the name of one *and only one* index level, use that level;
- otherwise it should be a number indicating level position.



## Examples

```
>>> idx = pd.Index([1,2,3])
>>> idx
Int64Index([1, 2, 3], dtype='int64')
```

Check whether each index value in a list of values. `>>> idx.isin([1, 4])` `array([ True, False, False])`

```
>>> midx = pd.MultiIndex.from_arrays([[1,2,3],
...                                  ['red', 'blue', 'green']],
...                                  names=('number', 'color'))
>>> midx
MultiIndex(levels=[[1, 2, 3], ['blue', 'green', 'red']],
            labels=[[0, 1, 2], [2, 0, 1]],
            names=['number', 'color'])
```

Check whether the strings in the ‘color’ level of the MultiIndex are in a list of colors.

```
>>> midx.isin(['red', 'orange', 'yellow'], level='color')
array([ True, False, False])
```

To check across the levels of a MultiIndex, pass a list of tuples:

```
>>> midx.isin([(1, 'red'), (3, 'red')])
array([ True, False, False])
```

For a DatetimeIndex, string values in *values* are converted to Timestamps.

```
>>> dates = ['2000-03-11', '2000-03-12', '2000-03-13']
>>> dti = pd.to_datetime(dates)
>>> dti
DatetimeIndex(['2000-03-11', '2000-03-12', '2000-03-13'],
              dtype='datetime64[ns]', freq=None)
```

```
>>> dti.isin(['2000-03-11'])
array([ True, False, False])
```

### 34.6.1.57 pandas.Index.isna

`Index.isna()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

New in version 0.20.0.

**Returns** `numpy.ndarray`

A boolean array of whether my values are NA

**See also:**

[`pandas.Index.notna`](#) boolean inverse of `isna`.

`pandas.Index.dropna` omit entries with missing values.

`pandas.isna` top-level isna.

`Series.isna` detect missing values in Series object.

## Examples

Show which entries in a `pandas.Index` are NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False,  True], dtype=bool)
```

Empty strings are not considered NA values. `None` is considered an NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False,  True], dtype=bool)
```

For datetimes, `NaT` (Not a Time) is considered as an NA value.

```
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
...                          pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
              dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False,  True,  True,  True], dtype=bool)
```

### 34.6.1.58 pandas.Index.isnull

`Index.isnull()`

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as `None`, `numpy.NaN` or `pd.NaT`, get mapped to `True` values. Everything else get mapped to `False` values. Characters such as empty strings `''` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`).

New in version 0.20.0.

#### Returns `numpy.ndarray`

A boolean array of whether my values are NA

See also:

`pandas.Index.notna` boolean inverse of `isna`.

`pandas.Index.dropna` omit entries with missing values.

`pandas.isna` top-level isna.

`Series.isna` detect missing values in Series object.

## Examples

Show which entries in a pandas.Index are NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.isna()
array([False, False,  True], dtype=bool)
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.isna()
array([False, False, False,  True], dtype=bool)
```

For datetimes, *NaT* (Not a Time) is considered as an NA value.

```
>>> idx = pd.DatetimeIndex([pd.Timestamp('1940-04-25'),
...                          pd.Timestamp(''), None, pd.NaT])
>>> idx
DatetimeIndex(['1940-04-25', 'NaT', 'NaT', 'NaT'],
              dtype='datetime64[ns]', freq=None)
>>> idx.isna()
array([False,  True,  True,  True], dtype=bool)
```

### 34.6.1.59 pandas.Index.item

`Index.item()`

return the first element of the underlying data as a python scalar

### 34.6.1.60 pandas.Index.join

`Index.join(other, how='left', level=None, return_indexers=False, sort=False)`  
*this is an internal non-public method*

Compute join\_index and indexers to conform data structures to the new index.

#### Parameters

**other** [Index]

**how** [{ 'left', 'right', 'inner', 'outer' }]

**level** [int or level name, default None]

**return\_indexers** [boolean, default False]

**sort** : boolean, default False

Sort the join keys lexicographically in the result Index. If False, the order of the join keys depends on the join type (how keyword)

New in version 0.20.0.

#### Returns

`join_index, (left_indexer, right_indexer)`

#### 34.6.1.61 `pandas.Index.map`

`Index.map` (*mapper*, *na\_action=None*)

Map values using input correspondence (a dict, Series, or function).

**Parameters** `mapper` : function, dict, or Series

Mapping correspondence.

`na_action` : {None, 'ignore'}

If 'ignore', propagate NA values, without passing them to the mapping correspondence.

**Returns** `applied` : Union[Index, MultiIndex], inferred

The output of the mapping function applied to the index. If the function returns a tuple with more than one element a MultiIndex will be returned.

#### 34.6.1.62 `pandas.Index.max`

`Index.max` ()

Return the maximum value of the Index.

**Returns** `scalar`

Maximum value.

**See also:**

[`Index.min`](#) Return the minimum value in an Index.

[`Series.max`](#) Return the maximum value in a Series.

[`DataFrame.max`](#) Return the maximum values in a DataFrame.

#### Examples

```
>>> idx = pd.Index([3, 2, 1])
>>> idx.max()
3
```

```
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.max()
'c'
```

For a MultiIndex, the maximum is determined lexicographically.

```
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.max()
('b', 2)
```

### 34.6.1.63 pandas.Index.memory\_usage

`Index.memory_usage (deep=False)`

Memory usage of the values

**Parameters** `deep` : bool

Introspect the data deeply, interrogate *object* dtypes for system-level memory consumption

**Returns**

bytes used

**See also:**

`numpy.ndarray.nbytes`

#### Notes

Memory usage does not include memory consumed by elements that are not components of the array if `deep=False` or if used on PyPy

### 34.6.1.64 pandas.Index.min

`Index.min()`

Return the minimum value of the Index.

**Returns** scalar

Minimum value.

**See also:**

**`Index.max`** Return the maximum value of the object.

**`Series.min`** Return the minimum value in a Series.

**`DataFrame.min`** Return the minimum values in a DataFrame.

#### Examples

```
>>> idx = pd.Index([3, 2, 1])
>>> idx.min()
1
```

```
>>> idx = pd.Index(['c', 'b', 'a'])
>>> idx.min()
'a'
```

For a MultiIndex, the minimum is determined lexicographically.

```
>>> idx = pd.MultiIndex.from_product([('a', 'b'), (2, 1)])
>>> idx.min()
('a', 1)
```

### 34.6.1.65 pandas.Index.notna

`Index.notna()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to `False` values.

New in version 0.20.0.

**Returns** `numpy.ndarray`

Boolean array to indicate which entries are not NA.

**See also:**

`Index.notnull` alias of `notna`

`Index.isna` inverse of `notna`

`pandas.notna` top-level `notna`

#### Examples

Show which entries in an `Index` are not NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True,  True, False])
```

Empty strings are not considered NA values. `None` is considered a NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True,  True,  True, False])
```

### 34.6.1.66 pandas.Index.notnull

`Index.notnull()`

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to `True`. Characters such as empty strings `' '` or `numpy.inf` are not considered NA values (unless you set `pandas.options.mode.use_inf_as_na = True`). NA values, such as `None` or `numpy.NaN`, get mapped to `False` values.

New in version 0.20.0.

**Returns** `numpy.ndarray`

Boolean array to indicate which entries are not NA.

**See also:**

`Index.notnull` alias of `notna`

`Index.isna` inverse of `notna`

`pandas.notna` top-level `notna`

## Examples

Show which entries in an Index are not NA. The result is an array.

```
>>> idx = pd.Index([5.2, 6.0, np.NaN])
>>> idx
Float64Index([5.2, 6.0, nan], dtype='float64')
>>> idx.notna()
array([ True,  True, False])
```

Empty strings are not considered NA values. None is considered a NA value.

```
>>> idx = pd.Index(['black', '', 'red', None])
>>> idx
Index(['black', '', 'red', None], dtype='object')
>>> idx.notna()
array([ True,  True,  True, False])
```

### 34.6.1.67 pandas.Index.nunique

`Index.nunique` (*dropna=True*)

Return number of unique elements in the object.

Excludes NA values by default.

**Parameters** `dropna` : boolean, default True

Don't include NaN in the count.

**Returns**

`nunique` [int]

### 34.6.1.68 pandas.Index.putmask

`Index.putmask` (*mask, value*)

return a new Index of the values set with the mask

**See also:**

`numpy.ndarray.putmask`

### 34.6.1.69 pandas.Index.ravel

`Index.ravel` (*order='C'*)

return an ndarray of the flattened values of the underlying data

**See also:**

`numpy.ndarray.ravel`

#### 34.6.1.70 pandas.Index.reindex

`Index.reindex` (*target*, *method=None*, *level=None*, *limit=None*, *tolerance=None*)  
Create index with target's values (move/add/delete values as necessary)

**Parameters**

**target** [an iterable]

**Returns** **new\_index** : `pd.Index`

Resulting index

**indexer** : `np.ndarray` or `None`

Indices of output values in original index

#### 34.6.1.71 pandas.Index.rename

`Index.rename` (*name*, *inplace=False*)  
Set new names on index. Defaults to returning new index.

**Parameters** **name** : str or list

name to set

**inplace** : bool

if True, mutates in place

**Returns**

new index (of same type and class...etc) [if inplace, returns None]

#### 34.6.1.72 pandas.Index.repeat

`Index.repeat` (*repeats*, *\*args*, *\*\*kwargs*)  
Repeat elements of an Index.

Returns a new index where each element of the current index is repeated consecutively a given number of times.

**Parameters** **repeats** : int

The number of repetitions for each element.

**\*\*kwargs**

Additional keywords have no effect but might be accepted for compatibility with numpy.

**Returns** **pandas.Index**

Newly created Index with repeated elements.

**See also:**

[`Series.repeat`](#) Equivalent function for Series

[`numpy.repeat`](#) Underlying implementation



## Examples

```
>>> idx = pd.Index([1, 2, 3])
>>> idx
Int64Index([1, 2, 3], dtype='int64')
>>> idx.repeat(2)
Int64Index([1, 1, 2, 2, 3, 3], dtype='int64')
>>> idx.repeat(3)
Int64Index([1, 1, 1, 2, 2, 2, 3, 3, 3], dtype='int64')
```

### 34.6.1.73 pandas.Index.searchsorted

`Index.searchsorted` (*value*, *side*='left', *sorter*=None)

Find indices where elements should be inserted to maintain order.

Find the indices into a sorted `IndexOpsMixin` *self* such that, if the corresponding elements in *value* were inserted before the indices, the order of *self* would be preserved.

**Parameters** *value* : array\_like

Values to insert into *self*.

**side** : {'left', 'right'}, optional

If 'left', the index of the first suitable location found is given. If 'right', return the last such index. If there is no suitable index, return either 0 or N (where N is the length of *self*).

**sorter** : 1-D array\_like, optional

Optional array of integer indices that sort *self* into ascending order. They are typically the result of `np.argsort`.

**Returns** *indices* : array of ints

Array of insertion points with the same shape as *value*.

**See also:**

`numpy.searchsorted`

## Notes

Binary search is used to find the required insertion points.

## Examples

```
>>> x = pd.Series([1, 2, 3])
>>> x
0    1
1    2
2    3
dtype: int64
```

```
>>> x.searchsorted(4)
array([3])
```

```
>>> x.searchsorted([0, 4])
array([0, 3])
```

```
>>> x.searchsorted([1, 3], side='left')
array([0, 2])
```

```
>>> x.searchsorted([1, 3], side='right')
array([1, 3])
```

```
>>> x = pd.Categorical(['apple', 'bread', 'bread',
                       'cheese', 'milk'], ordered=True)
[apple, bread, bread, cheese, milk]
Categories (4, object): [apple < bread < cheese < milk]
```

```
>>> x.searchsorted('bread')
array([1])      # Note: an array, not a scalar
```

```
>>> x.searchsorted(['bread'], side='right')
array([3])
```

### 34.6.1.74 pandas.Index.set\_names

`Index.set_names` (*names, level=None, inplace=False*)

Set new names on index. Defaults to returning new index.

**Parameters** **names** : str or sequence

name(s) to set

**level** : int, level name, or sequence of int/level names (default None)

If the index is a MultiIndex (hierarchical), level(s) to set (None for all levels).  
Otherwise level must be None

**inplace** : bool

if True, mutates in place

**Returns**

**new index (of same type and class...etc) [if inplace, returns None]**

### Examples

```
>>> Index([1, 2, 3, 4]).set_names('foo')
Int64Index([1, 2, 3, 4], dtype='int64', name='foo')
>>> Index([1, 2, 3, 4]).set_names(['foo'])
Int64Index([1, 2, 3, 4], dtype='int64', name='foo')
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                (2, u'one'), (2, u'two')],
                                names=['foo', 'bar'])
>>> idx.set_names(['baz', 'quz'])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=[u'baz', u'quz'])
```

(continues on next page)

(continued from previous page)

```
>>> idx.set_names('baz', level=0)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=[u'baz', u'bar'])
```

### 34.6.1.75 pandas.Index.set\_value

`Index.set_value(arr, key, value)`

Fast lookup of value from 1-dimensional ndarray. Only use this if you know what you're doing

### 34.6.1.76 pandas.Index.shift

`Index.shift(periods=1, freq=None)`

Shift index by desired number of time frequency increments.

This method is for shifting the values of datetime-like indexes by a specified time increment a given number of times.

**Parameters** `periods` : int, default 1

Number of periods (or increments) to shift by, can be positive or negative.

**freq** : pandas.DateOffset, pandas.Timedelta or string, optional

Frequency increment to shift by. If None, the index is shifted by its own *freq* attribute. Offset aliases are valid strings, e.g., 'D', 'W', 'M' etc.

**Returns** pandas.Index

shifted index

**See also:**

[`Series.shift`](#) Shift values of Series.

### Notes

This method is only implemented for datetime-like index classes, i.e., `DatetimeIndex`, `PeriodIndex` and `TimedeltaIndex`.

### Examples

Put the first 5 month starts of 2011 into an index.

```
>>> month_starts = pd.date_range('1/1/2011', periods=5, freq='MS')
>>> month_starts
DatetimeIndex(['2011-01-01', '2011-02-01', '2011-03-01', '2011-04-01',
              '2011-05-01'],
              dtype='datetime64[ns]', freq='MS')
```

Shift the index by 10 days.

```
>>> month_starts.shift(10, freq='D')
DatetimeIndex(['2011-01-11', '2011-02-11', '2011-03-11', '2011-04-11',
              '2011-05-11'],
              dtype='datetime64[ns]', freq=None)
```

The default value of *freq* is the *freq* attribute of the index, which is 'MS' (month start) in this example.

```
>>> month_starts.shift(10)
DatetimeIndex(['2011-11-01', '2011-12-01', '2012-01-01', '2012-02-01',
              '2012-03-01'],
              dtype='datetime64[ns]', freq='MS')
```

### 34.6.1.77 pandas.Index.slice\_indexer

`Index.slice_indexer` (*start=None, end=None, step=None, kind=None*)

For an ordered or unique index, compute the slice indexer for input labels and step.

**Parameters** **start** : label, default None

If None, defaults to the beginning

**end** : label, default None

If None, defaults to the end

**step** [int, default None]

**kind** [string, default None]

**Returns**

**indexer** [slice]

**Raises** **KeyError** : If key does not exist, or key is not unique and index is not ordered.

### Notes

This function assumes that the data is sorted, so use at your own peril

### Examples

This is a method on all index types. For example you can do:

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_indexer(start='b', end='c')
slice(1, 3)
```

```
>>> idx = pd.MultiIndex.from_arrays([list('abcd'), list('efgh')])
>>> idx.slice_indexer(start='b', end=('c', 'g'))
slice(1, 3)
```

### 34.6.1.78 pandas.Index.slice\_locs

`Index.slice_locs` (*start=None, end=None, step=None, kind=None*)

Compute slice locations for input labels.

**Parameters** **start** : label, default None

If None, defaults to the beginning

**end** : label, default None

If None, defaults to the end

**step** : int, defaults None

If None, defaults to 1

**kind** [{ 'ix', 'loc', 'getitem' } or None]

**Returns**

**start, end** [int]

**See also:**

[`Index.get\_loc`](#) Get location for a single label

#### Notes

This method only works if the index is monotonic or unique.

#### Examples

```
>>> idx = pd.Index(list('abcd'))
>>> idx.slice_locs(start='b', end='c')
(1, 3)
```

### 34.6.1.79 pandas.Index.sort\_values

`Index.sort_values` (*return\_indexer=False, ascending=True*)

Return a sorted copy of the index.

Return a sorted copy of the index, and optionally return the indices that sorted the index itself.

**Parameters** **return\_indexer** : bool, default False

Should the indices that would sort the index be returned.

**ascending** : bool, default True

Should the index values be sorted in an ascending order.

**Returns** **sorted\_index** : pandas.Index

Sorted copy of the index.

**indexer** : numpy.ndarray, optional

The indices that the index itself was sorted by.

See also:

`pandas.Series.sort_values` Sort values of a Series.

`pandas.DataFrame.sort_values` Sort values in a DataFrame.

### Examples

```
>>> idx = pd.Index([10, 100, 1, 1000])
>>> idx
Int64Index([10, 100, 1, 1000], dtype='int64')
```

Sort values in ascending order (default behavior).

```
>>> idx.sort_values()
Int64Index([1, 10, 100, 1000], dtype='int64')
```

Sort values in descending order, and also get the indices *idx* was sorted by.

```
>>> idx.sort_values(ascending=False, return_indexer=True)
(Int64Index([1000, 100, 10, 1], dtype='int64'), array([3, 1, 0, 2]))
```

#### 34.6.1.80 pandas.Index.sortlevel

`Index.sortlevel` (*level=None, ascending=True, sort\_remaining=None*)

For internal compatibility with with the Index API

Sort the Index. This is for compat with MultiIndex

**Parameters** `ascending` : boolean, default True

False to sort in descending order

`level, sort_remaining` are compat parameters

**Returns**

`sorted_index` [Index]

#### 34.6.1.81 pandas.Index.str

`Index.str()`

Vectorized string functions for Series and Index. NAs stay NA unless handled otherwise by a particular method. Patterned after Python's string methods, with some inspiration from R's stringr package.

### Examples

```
>>> s.str.split('_')
>>> s.str.replace('_', '')
```

### 34.6.1.82 pandas.Index.summary

`Index.summary` (*name=None*)

Return a summarized representation .. deprecated:: 0.23.0

### 34.6.1.83 pandas.Index.symmetric\_difference

`Index.symmetric_difference` (*other, result\_name=None*)

Compute the symmetric difference of two Index objects. It's sorted if sorting is possible.

#### Parameters

**other** [Index or array-like]

**result\_name** [str]

#### Returns

**symmetric\_difference** [Index]

#### Notes

`symmetric_difference` contains elements that appear in either `idx1` or `idx2` but not both. Equivalent to the Index created by `idx1.difference(idx2) | idx2.difference(idx1)` with duplicates dropped.

#### Examples

```
>>> idx1 = Index([1, 2, 3, 4])
>>> idx2 = Index([2, 3, 4, 5])
>>> idx1.symmetric_difference(idx2)
Int64Index([1, 5], dtype='int64')
```

You can also use the `^` operator:

```
>>> idx1 ^ idx2
Int64Index([1, 5], dtype='int64')
```

### 34.6.1.84 pandas.Index.take

`Index.take` (*indices, axis=0, allow\_fill=True, fill\_value=None, \*\*kwargs*)

return a new Index of the values selected by the indices

For internal compatibility with numpy arrays.

**Parameters** **indices** : list

Indices to be taken

**axis** : int, optional

The axis over which to select values, always 0.

**allow\_fill** [bool, default True]

**fill\_value** : bool, default None

If `allow_fill=True` and `fill_value` is not None, indices specified by -1 is regarded as NA. If Index doesn't hold NA, raise `ValueError`

**See also:**

`numpy.ndarray.take`

### 34.6.1.85 `pandas.Index.to_frame`

`Index.to_frame(index=True)`

Create a `DataFrame` with a column containing the Index.

New in version 0.21.0.

**Parameters** `index` : boolean, default True

Set the index of the returned `DataFrame` as the original Index.

**Returns** `DataFrame`

`DataFrame` containing the original Index data.

**See also:**

[`Index.to\_series`](#) Convert an Index to a Series.

[`Series.to\_frame`](#) Convert Series to `DataFrame`.

### Examples

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
      animal
animal
Ant      Ant
Bear    Bear
Cow     Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
      animal
0      Ant
1     Bear
2      Cow
```

### 34.6.1.86 `pandas.Index.to_native_types`

`Index.to_native_types(slicer=None, **kwargs)`

Format specified values of *self* and return them.

**Parameters** `slicer` : int, array-like

An indexer into *self* that specifies which values are used in the formatting process.

**kwargs** : dict



Options for specifying how the values should be formatted. These options include the following:

1. **na\_rep** [str] The value that serves as a placeholder for NULL values
2. **quoting** [bool or None] Whether or not there are quoted values in *self*
3. **date\_format** [str] The format used to represent date-like values

### 34.6.1.87 pandas.Index.to\_series

`Index.to_series (index=None, name=None)`

Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index

**Parameters** **index** : Index, optional

index of resulting Series. If None, defaults to original index

**name** : string, optional

name of resulting Series. If None, defaults to name of original index

**Returns**

**Series** [dtype will be based on the type of the Index values.]

### 34.6.1.88 pandas.Index.tolist

`Index.tolist ()`

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

**See also:**

`numpy.ndarray.tolist`

### 34.6.1.89 pandas.Index.transpose

`Index.transpose (*args, **kwargs)`

return the transpose, which is by definition self

### 34.6.1.90 pandas.Index.union

`Index.union (other)`

Form the union of two Index objects and sorts if possible.

**Parameters**

**other** [Index or array-like]

**Returns**

**union** [Index]

## Examples

```
>>> idx1 = pd.Index([1, 2, 3, 4])
>>> idx2 = pd.Index([3, 4, 5, 6])
>>> idx1.union(idx2)
Int64Index([1, 2, 3, 4, 5, 6], dtype='int64')
```

### 34.6.1.91 pandas.Index.unique

`Index.unique` (*level=None*)

Return unique values in the index. Uniques are returned in order of appearance, this does NOT sort.

**Parameters** **level** : int or str, optional, default None

Only return values from specified level (for MultiIndex)

New in version 0.23.0.

**Returns**

**Index without duplicates**

**See also:**

*unique*, *Series.unique*

### 34.6.1.92 pandas.Index.value\_counts

`Index.value_counts` (*normalize=False, sort=True, ascending=False, bins=None, dropna=True*)

Returns object containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

**Parameters** **normalize** : boolean, default False

If True then the object returned will contain the relative frequencies of the unique values.

**sort** : boolean, default True

Sort by values

**ascending** : boolean, default False

Sort in ascending order

**bins** : integer, optional

Rather than count values, group them into half-open bins, a convenience for `pd.cut`, only works with numeric data

**dropna** : boolean, default True

Don't include counts of NaN.

**Returns**

**counts** [Series]

### 34.6.1.93 pandas.Index.where

`Index.where` (*cond*, *other=None*)

New in version 0.19.0.

Return an Index of same shape as self and whose corresponding entries are from self where *cond* is True and otherwise are from *other*.

#### Parameters

**cond** [boolean array-like with the same length as self]

**other** [scalar, or array-like]

<b>holds_integer</b>	
<b>is_boolean</b>	
<b>is_floating</b>	
<b>is_integer</b>	
<b>is_interval</b>	
<b>is_lexsorted_for_tuple</b>	
<b>is_mixed</b>	
<b>is_numeric</b>	
<b>is_object</b>	
<b>is_type_compatible</b>	
<b>sort</b>	
<b>view</b>	

## 34.6.2 Attributes

<code>Index.values</code>	return the underlying data as an ndarray
<code>Index.is_monotonic</code>	alias for <code>is_monotonic_increasing</code> (deprecated)
<code>Index.is_monotonic_increasing</code>	return if the index is monotonic increasing (only equal or increasing) values.
<code>Index.is_monotonic_decreasing</code>	return if the index is monotonic decreasing (only equal or decreasing) values.
<code>Index.is_unique</code>	return if the index has unique values
<code>Index.has_duplicates</code>	
<code>Index.hasnans</code>	return if I have any nans; enables various perf speedups
<code>Index.dtype</code>	return the dtype object of the underlying data
<code>Index.dtype_str</code>	return the dtype str of the underlying data
<code>Index.inferred_type</code>	return a string of the type inferred from the values
<code>Index.is_all_dates</code>	
<code>Index.shape</code>	return a tuple of the shape of the underlying data
<code>Index.name</code>	
<code>Index.names</code>	
<code>Index.nbytes</code>	return the number of bytes in the underlying data
<code>Index.ndim</code>	return the number of dimensions of the underlying data, by definition 1
<code>Index.size</code>	return the number of elements in the underlying data
<code>Index.empty</code>	
<code>Index.strides</code>	return the strides of the underlying data

Continued on next page

Table 98 – continued from previous page

<code>Index.itemsize</code>	return the size of the dtype of the item of the underlying data
<code>Index.base</code>	return the base object if the memory of the underlying data is shared
<code>Index.T</code>	return the transpose, which is by definition self
<code>Index.memory_usage([deep])</code>	Memory usage of the values

**34.6.2.1 pandas.Index.has\_duplicates**`Index.has_duplicates`**34.6.2.2 pandas.Index.is\_all\_dates**`Index.is_all_dates`**34.6.2.3 pandas.Index.name**`Index.name = None`**34.6.2.4 pandas.Index.names**`Index.names`**34.6.2.5 pandas.Index.empty**`Index.empty`**34.6.3 Modifying and Computations**

<code>Index.all(*args, **kwargs)</code>	Return whether all elements are True.
<code>Index.any(*args, **kwargs)</code>	Return whether any element is True.
<code>Index.argmin([axis])</code>	return a ndarray of the minimum argument indexer
<code>Index.argmax([axis])</code>	return a ndarray of the maximum argument indexer
<code>Index.copy([name, deep, dtype])</code>	Make a copy of this object.
<code>Index.delete(loc)</code>	Make new Index with passed location(-s) deleted
<code>Index.drop(labels[, errors])</code>	Make new Index with passed list of labels deleted
<code>Index.drop_duplicates([keep])</code>	Return Index with duplicate values removed.
<code>Index.duplicated([keep])</code>	Indicate duplicate index values.
<code>Index.equals(other)</code>	Determines if two Index objects contain the same elements.
<code>Index.factorize([sort, na_sentinel])</code>	Encode the object as an enumerated type or categorical variable.
<code>Index.identical(other)</code>	Similar to equals, but check that other comparable attributes are also equal
<code>Index.insert(loc, item)</code>	Make new Index inserting new item at location.
<code>Index.is_(other)</code>	More flexible, faster check like <code>is</code> but that works through views

Continued on next page

Table 99 – continued from previous page

<i>Index.is_boolean()</i>	
<i>Index.is_categorical()</i>	Check if the Index holds categorical data.
<i>Index.is_floating()</i>	
<i>Index.is_integer()</i>	
<i>Index.is_interval()</i>	
<i>Index.is_lexsorted_for_tuple(tup)</i>	
<i>Index.is_mixed()</i>	
<i>Index.is_numeric()</i>	
<i>Index.is_object()</i>	
<i>Index.min()</i>	Return the minimum value of the Index.
<i>Index.max()</i>	Return the maximum value of the Index.
<i>Index.reindex(target[, method, level, ...])</i>	Create index with target's values (move/add/delete values as necessary)
<i>Index.rename(name[, inplace])</i>	Set new names on index.
<i>Index.repeat(repeats, *args, **kwargs)</i>	Repeat elements of an Index.
<i>Index.where(cond[, other])</i>	
<i>Index.take(indices[, axis, allow_fill, ...])</i>	return a new Index of the values selected by the indices
<i>Index.putmask(mask, value)</i>	return a new Index of the values set with the mask
<i>Index.set_names(names[, level, inplace])</i>	Set new names on index.
<i>Index.unique([level])</i>	Return unique values in the index.
<i>Index.nunique([dropna])</i>	Return number of unique elements in the object.
<i>Index.value_counts([normalize, sort, ...])</i>	Returns object containing counts of unique values.

**34.6.3.1 pandas.Index.is\_boolean**`Index.is_boolean()`**34.6.3.2 pandas.Index.is\_floating**`Index.is_floating()`**34.6.3.3 pandas.Index.is\_integer**`Index.is_integer()`**34.6.3.4 pandas.Index.is\_interval**`Index.is_interval()`**34.6.3.5 pandas.Index.is\_lexsorted\_for\_tuple**`Index.is_lexsorted_for_tuple(tup)`**34.6.3.6 pandas.Index.is\_mixed**`Index.is_mixed()`

### 34.6.3.7 pandas.Index.is\_numeric

`Index.is_numeric()`

### 34.6.3.8 pandas.Index.is\_object

`Index.is_object()`

## 34.6.4 Missing Values

<code>Index.fillna([value, downcast])</code>	Fill NA/NaN values with the specified value
<code>Index.dropna([how])</code>	Return Index without NA/NaN values
<code>Index.isna()</code>	Detect missing values.
<code>Index.notna()</code>	Detect existing (non-missing) values.

## 34.6.5 Conversion

<code>Index.astype(dtype[, copy])</code>	Create an Index with values cast to dtypes.
<code>Index.item()</code>	return the first element of the underlying data as a python scalar
<code>Index.map(mapper[, na_action])</code>	Map values using input correspondence (a dict, Series, or function).
<code>Index.ravel([order])</code>	return an ndarray of the flattened values of the underlying data
<code>Index.tolist()</code>	Return a list of the values.
<code>Index.to_native_types([slicer])</code>	Format specified values of <i>self</i> and return them.
<code>Index.to_series([index, name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
<code>Index.to_frame([index])</code>	Create a DataFrame with a column containing the Index.
<code>Index.view([cls])</code>	

### 34.6.5.1 pandas.Index.view

`Index.view(cls=None)`

## 34.6.6 Sorting

<code>Index.argsort(*args, **kwargs)</code>	Return the integer indicies that would sort the index.
<code>Index.searchsorted(value[, side, sorter])</code>	Find indices where elements should be inserted to maintain order.
<code>Index.sort_values([return_indexer, ascending])</code>	Return a sorted copy of the index.

## 34.6.7 Time-specific operations

<code>Index.shift([periods, freq])</code>	Shift index by desired number of time frequency increments.
---	---

### 34.6.8 Combining / joining / set operations

<code>Index.append(other)</code>	Append a collection of Index options together
<code>Index.join(other[, how, level, ...])</code>	<i>this is an internal non-public method</i>
<code>Index.intersection(other)</code>	Form the intersection of two Index objects.
<code>Index.union(other)</code>	Form the union of two Index objects and sorts if possible.
<code>Index.difference(other)</code>	Return a new Index with elements from the index that are not in <i>other</i> .
<code>Index.symmetric_difference(other[, re-sult_name])</code>	Compute the symmetric difference of two Index objects.

### 34.6.9 Selecting

<code>Index.asof(label)</code>	For a sorted index, return the most recent label up to and including the passed label.
<code>Index.asof_locs(where, mask)</code>	where : array of timestamps mask : array of booleans where data is not NA
<code>Index.contains(key)</code>	return a boolean if this key is IN the index
<code>Index.get_duplicates()</code>	(DEPRECATED) Extract duplicated index elements.
<code>Index.get_indexer(target[, method, limit, ...])</code>	Compute indexer and mask for new index given the current index.
<code>Index.get_indexer_for(target, **kwargs)</code>	guaranteed return of an indexer even when non-unique This dispatches to <code>get_indexer</code> or <code>get_indexer_nonunique</code> as appropriate
<code>Index.get_indexer_non_unique(target)</code>	Compute indexer and mask for new index given the current index.
<code>Index.get_level_values(level)</code>	Return an Index of values for requested level, equal to the length of the index.
<code>Index.get_loc(key[, method, tolerance])</code>	Get integer location, slice or boolean mask for requested label.
<code>Index.get_slice_bound(label, side, kind)</code>	Calculate slice bound that corresponds to given label.
<code>Index.get_value(series, key)</code>	Fast lookup of value from 1-dimensional ndarray.
<code>Index.get_values()</code>	Return <i>Index</i> data as a <i>numpy.ndarray</i> .
<code>Index.set_value(arr, key, value)</code>	Fast lookup of value from 1-dimensional ndarray.
<code>Index.isin(values[, level])</code>	Return a boolean array where the index values are in <i>values</i> .
<code>Index.slice_indexer([start, end, step, kind])</code>	For an ordered or unique index, compute the slice indexer for input labels and step.
<code>Index.slice_locs([start, end, step, kind])</code>	Compute slice locations for input labels.

## 34.7 Numeric Index

<i>RangeIndex</i>	Immutable Index implementing a monotonic integer range.
<i>Int64Index</i>	Immutable ndarray implementing an ordered, sliceable set.
<i>UInt64Index</i>	Immutable ndarray implementing an ordered, sliceable set.
<i>Float64Index</i>	Immutable ndarray implementing an ordered, sliceable set.

### 34.7.1 pandas.RangeIndex

**class** pandas.RangeIndex

Immutable Index implementing a monotonic integer range.

RangeIndex is a memory-saving special case of Int64Index limited to representing monotonic ranges. Using RangeIndex may in some instances improve computing speed.

This is the default index type used by DataFrame and Series when no explicit index is provided by the user.

**Parameters** **start** : int (default: 0), or other RangeIndex instance.

If int and “stop” is not given, interpreted as “stop” instead.

**stop** [int (default: 0)]

**step** [int (default: 1)]

**name** : object, optional

Name to be stored in the index

**copy** : bool, default False

Unused, accepted for homogeneity with other index types.

**See also:**

**Index** The base pandas Index type

**Int64Index** Index of int64 data

#### Attributes

None	
------	--

#### Methods

<i>from_range</i> (data[, name, dtype])	create RangeIndex from a range (py3), or xrange (py2) object
---	--



### 34.7.1.1 pandas.RangeIndex.from\_range

**classmethod** `RangeIndex.from_range` (*data*, *name=None*, *dtype=None*, *\*\*kwargs*)  
 create RangeIndex from a range (py3), or xrange (py2) object

### 34.7.2 pandas.Int64Index

**class** `pandas.Int64Index`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Int64Index is a special case of *Index* with purely integer labels.

#### Parameters

**data** [array-like (1-dimensional)]

**dtype** [NumPy dtype (default: int64)]

**copy** : bool

Make a copy of input ndarray

**name** : object

Name to be stored in the index

See also:

[\*Index\*](#) The base pandas Index type

#### Notes

An Index instance can **only** contain hashable objects.

#### Attributes

None	
------	--

#### Methods

None	
------	--

### 34.7.3 pandas.UInt64Index

**class** `pandas.UInt64Index`

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. UInt64Index is a special case of *Index* with purely unsigned integer labels.

#### Parameters

**data** [array-like (1-dimensional)]

**dtype** [NumPy dtype (default: uint64)]

**copy** : bool

Make a copy of input ndarray

**name** : object

Name to be stored in the index

**See also:**

[\*Index\*](#) The base pandas Index type

### Notes

An Index instance can **only** contain hashable objects.

### Attributes

None	
------	--

### Methods

None	
------	--

## 34.7.4 pandas.Float64Index

**class** pandas.**Float64Index**

Immutable ndarray implementing an ordered, sliceable set. The basic object storing axis labels for all pandas objects. Float64Index is a special case of *Index* with purely float labels.

### Parameters

**data** [array-like (1-dimensional)]

**dtype** [NumPy dtype (default: float64)]

**copy** : bool

Make a copy of input ndarray

**name** : object

Name to be stored in the index

**See also:**

[\*Index\*](#) The base pandas Index type

### Notes

An Index instance can **only** contain hashable objects.

Attributes

None	
------	--

Methods

None	
------	--

<i>RangeIndex.from_range</i> (data[, name, dtype])	create RangeIndex from a range (py3), or xrange (py2) object
--	--

34.8 CategoricalIndex

<i>CategoricalIndex</i>	Immutable Index implementing an ordered, sliceable set.
-------------------------	---

34.8.1 pandas.CategoricalIndex

**class** pandas.CategoricalIndex  
Immutable Index implementing an ordered, sliceable set. CategoricalIndex represents a sparsely populated Index with an underlying Categorical.

Parameters

- data** [array-like or Categorical, (1-dimensional)]
- categories** : optional, array-like  
categories for the CategoricalIndex
- ordered** : boolean,  
designating if the categories are ordered
- copy** : bool  
Make a copy of input ndarray
- name** : object  
Name to be stored in the index

See also:

*Categorical*, *Index*

Attributes

<b>codes</b>	
<b>categories</b>	
<b>ordered</b>	

## Methods

<code>rename_categories(*args, **kwargs)</code>	Renames categories.
<code>reorder_categories(*args, **kwargs)</code>	Reorders categories as specified in <code>new_categories</code> .
<code>add_categories(*args, **kwargs)</code>	Add new categories.
<code>remove_categories(*args, **kwargs)</code>	Removes the specified categories.
<code>remove_unused_categories(*args, **kwargs)</code>	Removes categories which are not used.
<code>set_categories(*args, **kwargs)</code>	Sets the categories to the specified <code>new_categories</code> .
<code>as_ordered(*args, **kwargs)</code>	Sets the Categorical to be ordered
<code>as_unordered(*args, **kwargs)</code>	Sets the Categorical to be unordered
<code>map(mapper)</code>	Map values using input correspondence (a dict, Series, or function).

### 34.8.1.1 pandas.CategoricalIndex.rename\_categories

`CategoricalIndex.rename_categories(*args, **kwargs)`

Renames categories.

**Parameters** `new_categories` : list-like, dict-like or callable

- list-like: all items must be unique and the number of items in the new categories must match the existing number of categories.
- dict-like: specifies a mapping from old categories to new. Categories not contained in the mapping are passed through and extra categories in the mapping are ignored.

New in version 0.21.0.

- callable : a callable that is called on all items in the old categories and whose return values comprise the new categories.

New in version 0.23.0.

**Warning:** Currently, Series are considered list like. In a future version of pandas they'll be considered dict-like.

**inplace** : boolean (default: False)

Whether or not to rename the categories inplace or return a copy of this categorical with renamed categories.

**Returns** `cat` : Categorical or None

With `inplace=False`, the new categorical is returned. With `inplace=True`, there is no return value.

**Raises** `ValueError`

If new categories are list-like and do not have the same number of items than the current categories or do not validate as categories

**See also:**

`reorder_categories`, `add_categories`, `remove_categories`,  
`remove_unused_categories`, `set_categories`

## Examples

```
>>> c = Categorical(['a', 'a', 'b'])
>>> c.rename_categories([0, 1])
[0, 0, 1]
Categories (2, int64): [0, 1]
```

For dict-like `new_categories`, extra keys are ignored and categories not in the dictionary are passed through

```
>>> c.rename_categories({'a': 'A', 'c': 'C'})
[A, A, b]
Categories (2, object): [A, b]
```

You may also provide a callable to create the new categories

```
>>> c.rename_categories(lambda x: x.upper())
[A, A, B]
Categories (2, object): [A, B]
```

### 34.8.1.2 pandas.CategoricalIndex.reorder\_categories

`CategoricalIndex.reorder_categories(*args, **kwargs)`

Reorders categories as specified in `new_categories`.

`new_categories` need to include all old categories and no new category items.

**Parameters** `new_categories` : Index-like

The categories in new order.

**ordered** : boolean, optional

Whether or not the categorical is treated as a ordered categorical. If not given, do not change the ordered information.

**inplace** : boolean (default: False)

Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

**cat** [Categorical with reordered categories or None if inplace.]

**Raises** `ValueError`

If the new categories do not contain all old category items or any new ones

**See also:**

`rename_categories`, `add_categories`, `remove_categories`,  
`remove_unused_categories`, `set_categories`

### 34.8.1.3 pandas.CategoricalIndex.add\_categories

`CategoricalIndex.add_categories(*args, **kwargs)`

Add new categories.

*new\_categories* will be included at the last/highest place in the categories and will be unused directly after this call.

**Parameters** **new\_categories** : category or list-like of category

The new categories to be included.

**inplace** : boolean (default: False)

Whether or not to add the categories inplace or return a copy of this categorical with added categories.

**Returns**

**cat** [Categorical with new categories added or None if inplace.]

**Raises** **ValueError**

If the new categories include old categories or do not validate as categories

**See also:**

*rename\_categories*, *reorder\_categories*, *remove\_categories*,  
*remove\_unused\_categories*, *set\_categories*

#### 34.8.1.4 pandas.CategoricalIndex.remove\_categories

`CategoricalIndex.remove_categories(*args, **kwargs)`

Removes the specified categories.

*removals* must be included in the old categories. Values which were in the removed categories will be set to NaN

**Parameters** **removals** : category or list of categories

The categories which should be removed.

**inplace** : boolean (default: False)

Whether or not to remove the categories inplace or return a copy of this categorical with removed categories.

**Returns**

**cat** [Categorical with removed categories or None if inplace.]

**Raises** **ValueError**

If the removals are not contained in the categories

**See also:**

*rename\_categories*, *reorder\_categories*, *add\_categories*,  
*remove\_unused\_categories*, *set\_categories*

#### 34.8.1.5 pandas.CategoricalIndex.remove\_unused\_categories

`CategoricalIndex.remove_unused_categories(*args, **kwargs)`

Removes categories which are not used.

**Parameters** **inplace** : boolean (default: False)

Whether or not to drop unused categories inplace or return a copy of this categorical with unused categories dropped.

**Returns**

**cat** [Categorical with unused categories dropped or None if inplace.]

**See also:**

*rename\_categories*, *reorder\_categories*, *add\_categories*, *remove\_categories*, *set\_categories*

**34.8.1.6 pandas.CategoricalIndex.set\_categories**

`CategoricalIndex.set_categories(*args, **kwargs)`

Sets the categories to the specified new\_categories.

*new\_categories* can include new categories (which will result in unused categories) or remove old categories (which results in values set to NaN). If *rename==True*, the categories will simply be renamed (less or more items than in old categories will result in values set to NaN or in unused categories respectively).

This method can be used to perform more than one action of adding, removing, and reordering simultaneously and is therefore faster than performing the individual steps via the more specialised methods.

On the other hand this method does not do checks (e.g., whether the old categories are included in the new categories on a reorder), which can result in surprising changes, for example when using special string dtypes on python3, which does not consider a S1 string equal to a single char python string.

**Parameters** *new\_categories* : Index-like

The categories in new order.

**ordered** : boolean, (default: False)

Whether or not the categorical is treated as an ordered categorical. If not given, do not change the ordered information.

**rename** : boolean (default: False)

Whether or not the *new\_categories* should be considered as a rename of the old categories or as reordered categories.

**inplace** : boolean (default: False)

Whether or not to reorder the categories inplace or return a copy of this categorical with reordered categories.

**Returns**

**cat** [Categorical with reordered categories or None if inplace.]

**Raises** `ValueError`

If *new\_categories* does not validate as categories

**See also:**

*rename\_categories*, *reorder\_categories*, *add\_categories*, *remove\_categories*, *remove\_unused\_categories*

**34.8.1.7 pandas.CategoricalIndex.as\_ordered**

`CategoricalIndex.as_ordered(*args, **kwargs)`

Sets the Categorical to be ordered

**Parameters** *inplace* : boolean (default: False)

Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to True

#### 34.8.1.8 pandas.CategoricalIndex.as\_unordered

`CategoricalIndex.as_unordered(*args, **kwargs)`

Sets the Categorical to be unordered

**Parameters** `inplace` : boolean (default: False)

Whether or not to set the ordered attribute inplace or return a copy of this categorical with ordered set to False

#### 34.8.1.9 pandas.CategoricalIndex.map

`CategoricalIndex.map(mapper)`

Map values using input correspondence (a dict, Series, or function).

Maps the values (their categories, not the codes) of the index to new categories. If the mapping correspondence is one-to-one the result is a `CategoricalIndex` which has the same order property as the original, otherwise an `Index` is returned.

If a `dict` or `Series` is used any unmapped category is mapped to `NaN`. Note that if this happens an `Index` will be returned.

**Parameters** `mapper` : function, dict, or Series

Mapping correspondence.

**Returns** `pandas.CategoricalIndex` or `pandas.Index`

Mapped index.

See also:

**`Index.map`** Apply a mapping correspondence on an `Index`.

**`Series.map`** Apply a mapping correspondence on a `Series`.

**`Series.apply`** Apply more complex functions on a `Series`.

#### Examples

```
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'])
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'],
                  ordered=False, dtype='category')
>>> idx.map(lambda x: x.upper())
CategoricalIndex(['A', 'B', 'C'], categories=['A', 'B', 'C'],
                  ordered=False, dtype='category')
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'third'})
CategoricalIndex(['first', 'second', 'third'], categories=['first',
                  'second', 'third'], ordered=False, dtype='category')
```

If the mapping is one-to-one the ordering of the categories is preserved:



```
>>> idx = pd.CategoricalIndex(['a', 'b', 'c'], ordered=True)
>>> idx
CategoricalIndex(['a', 'b', 'c'], categories=['a', 'b', 'c'],
                  ordered=True, dtype='category')
>>> idx.map({'a': 3, 'b': 2, 'c': 1})
CategoricalIndex([3, 2, 1], categories=[3, 2, 1], ordered=True,
                  dtype='category')
```

If the mapping is not one-to-one an *Index* is returned:

```
>>> idx.map({'a': 'first', 'b': 'second', 'c': 'first'})
Index(['first', 'second', 'first'], dtype='object')
```

If a *dict* is used, all unmapped categories are mapped to *NaN* and the result is an *Index*:

```
>>> idx.map({'a': 'first', 'b': 'second'})
Index(['first', 'second', nan], dtype='object')
```

## 34.8.2 Categorical Components

<i>CategoricalIndex.codes</i>	
<i>CategoricalIndex.categories</i>	
<i>CategoricalIndex.ordered</i>	
<i>CategoricalIndex.rename_categories(*args, ...)</i>	Renames categories.
<i>CategoricalIndex.reorder_categories(*args, ...)</i>	Reorders categories as specified in <i>new_categories</i> .
<i>CategoricalIndex.add_categories(*args, **kwargs)</i>	Add new categories.
<i>CategoricalIndex.remove_categories(*args, ...)</i>	Removes the specified categories.
<i>CategoricalIndex.remove_unused_categories(...)</i>	Removes categories which are not used.
<i>CategoricalIndex.set_categories(*args, **kwargs)</i>	Sets the categories to the specified <i>new_categories</i> .
<i>CategoricalIndex.as_ordered(*args, **kwargs)</i>	Sets the Categorical to be ordered
<i>CategoricalIndex.as_unordered(*args, **kwargs)</i>	Sets the Categorical to be unordered
<i>CategoricalIndex.map(mapper)</i>	Map values using input correspondence (a dict, Series, or function).

### 34.8.2.1 pandas.CategoricalIndex.codes

*CategoricalIndex.codes*

### 34.8.2.2 pandas.CategoricalIndex.categories

*CategoricalIndex.categories*

### 34.8.2.3 pandas.CategoricalIndex.ordered

CategoricalIndex.ordered

## 34.9 IntervalIndex

---

*IntervalIndex*

Immutable Index implementing an ordered, sliceable set.

---

### 34.9.1 pandas.IntervalIndex

**class** pandas.IntervalIndex

Immutable Index implementing an ordered, sliceable set. IntervalIndex represents an Index of Interval objects that are all closed on the same side.

New in version 0.20.0.

**Warning:** The indexing behaviors are provisional and may change in a future version of pandas.

**Parameters** **data** : array-like (1-dimensional)

Array-like containing Interval objects from which to build the IntervalIndex

**closed** : { 'left', 'right', 'both', 'neither' }, default 'right'

Whether the intervals are closed on the left-side, right-side, both or neither.

**name** : object, optional

Name to be stored in the index.

**copy** : boolean, default False

Copy the meta-data

**dtype** : dtype or None, default None

If None, dtype will be inferred

New in version 0.23.0.

**See also:**

**Index** The base pandas Index type

**Interval** A bounded slice-like interval; the elements of an IntervalIndex

**interval\_range** Function to create a fixed frequency IntervalIndex

*cut, qcut*

### Notes

See the [user guide](#) for more.

## Examples

A new `IntervalIndex` is typically constructed using `interval_range()`:

```
>>> pd.interval_range(start=0, end=5)
IntervalIndex([(0, 1], (1, 2], (2, 3], (3, 4], (4, 5]]
              closed='right', dtype='interval[int64]')
```

It may also be constructed using one of the constructor methods: `IntervalIndex.from_arrays()`, `IntervalIndex.from_breaks()`, and `IntervalIndex.from_tuples()`.

See further examples in the doc strings of `interval_range` and the mentioned constructor methods.

## Attributes

<code>closed</code>	Whether the intervals are closed on the left-side, right-side, both or neither
<code>is_non_overlapping_monotonic</code>	Return True if the <code>IntervalIndex</code> is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False
<code>left</code>	Return the left endpoints of each Interval in the <code>IntervalIndex</code> as an Index
<code>length</code>	Return an Index with entries denoting the length of each Interval in the <code>IntervalIndex</code>
<code>mid</code>	Return the midpoint of each Interval in the <code>IntervalIndex</code> as an Index
<code>right</code>	Return the right endpoints of each Interval in the <code>IntervalIndex</code> as an Index
<code>values</code>	Return the <code>IntervalIndex</code> 's data as a numpy array of Interval objects (with dtype='object')

### 34.9.1.1 pandas.IntervalIndex.closed

`IntervalIndex.closed`

Whether the intervals are closed on the left-side, right-side, both or neither

### 34.9.1.2 pandas.IntervalIndex.is\_non\_overlapping\_monotonic

`IntervalIndex.is_non_overlapping_monotonic`

Return True if the `IntervalIndex` is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False

### 34.9.1.3 pandas.IntervalIndex.left

`IntervalIndex.left`

Return the left endpoints of each Interval in the `IntervalIndex` as an Index

#### 34.9.1.4 pandas.IntervalIndex.length

`IntervalIndex.length`

Return an Index with entries denoting the length of each Interval in the IntervalIndex

#### 34.9.1.5 pandas.IntervalIndex.mid

`IntervalIndex.mid`

Return the midpoint of each Interval in the IntervalIndex as an Index

#### 34.9.1.6 pandas.IntervalIndex.right

`IntervalIndex.right`

Return the right endpoints of each Interval in the IntervalIndex as an Index

#### 34.9.1.7 pandas.IntervalIndex.values

`IntervalIndex.values`

Return the IntervalIndex's data as a numpy array of Interval objects (with dtype='object')

### Methods

<code>contains(key)</code>	Return a boolean indicating if the key is IN the index
<code>from_arrays(left, right[, closed, name, ...])</code>	Construct from two arrays defining the left and right bounds.
<code>from_breaks(breaks[, closed, name, copy, dtype])</code>	Construct an IntervalIndex from an array of splits
<code>from_tuples(data[, closed, name, copy, dtype])</code>	Construct an IntervalIndex from a list/array of tuples
<code>get_indexer(target[, method, limit, tolerance])</code>	Compute indexer and mask for new index given the current index.
<code>get_loc(key[, method])</code>	Get integer location, slice or boolean mask for requested label.

#### 34.9.1.8 pandas.IntervalIndex.contains

`IntervalIndex.contains(key)`

Return a boolean indicating if the key is IN the index

We accept / allow keys to be not *just* actual objects.

##### Parameters

**key** [int, float, Interval]

##### Returns

**boolean**

### 34.9.1.9 pandas.IntervalIndex.from\_arrays

**classmethod** `IntervalIndex.from_arrays` (*left*, *right*, *closed*='right', *name*=None, *copy*=False, *dtype*=None)

Construct from two arrays defining the left and right bounds.

**Parameters** *left* : array-like (1-dimensional)

Left bounds for each interval.

*right* : array-like (1-dimensional)

Right bounds for each interval.

*closed* : {'left', 'right', 'both', 'neither'}, default 'right'

Whether the intervals are closed on the left-side, right-side, both or neither.

*name* : object, optional

Name to be stored in the index.

*copy* : boolean, default False

Copy the data.

*dtype* : dtype, optional

If None, dtype will be inferred.

New in version 0.23.0.

#### Returns

**index** [IntervalIndex]

#### Raises ValueError

When a value is missing in only one of *left* or *right*. When a value in *left* is greater than the corresponding value in *right*.

See also:

[\*interval\\_range\*](#) Function to create a fixed frequency IntervalIndex.

[\*IntervalIndex.from\\_breaks\*](#) Construct an IntervalIndex from an array of splits.

[\*IntervalIndex.from\\_tuples\*](#) Construct an IntervalIndex from a list/array of tuples.

#### Notes

Each element of *left* must be less than or equal to the *right* element at the same position. If an element is missing, it must be missing in both *left* and *right*. A `TypeError` is raised when using an unsupported type for *left* or *right*. At the moment, 'category', 'object', and 'string' subtypes are not supported.

#### Examples

```
>>> pd.IntervalIndex.from_arrays([0, 1, 2], [1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3])
              closed='right',
              dtype='interval[int64]')
```

If you want to segment different groups of people based on ages, you can apply the method as follows:

```
>>> ages = pd.IntervalIndex.from_arrays([0, 2, 13],
...                                     [2, 13, 19], closed='left')
>>> ages
IntervalIndex([[0, 2), [2, 13), [13, 19)]
              closed='left',
              dtype='interval[int64]')
>>> s = pd.Series(['baby', 'kid', 'teen'], ages)
>>> s
[0, 2)      baby
[2, 13)     kid
[13, 19)    teen
dtype: object
```

Values may be missing, but they must be missing in both arrays.

```
>>> pd.IntervalIndex.from_arrays([0, np.nan, 13],
...                               [2, np.nan, 19])
IntervalIndex([(0.0, 2.0], nan, (13.0, 19.0])
              closed='right',
              dtype='interval[float64]')
```

#### 34.9.1.10 pandas.IntervalIndex.from\_breaks

**classmethod** `IntervalIndex.from_breaks` (*breaks*, *closed='right'*, *name=None*, *copy=False*, *dtype=None*)

Construct an IntervalIndex from an array of splits

**Parameters** *breaks* : array-like (1-dimensional)

Left and right bounds for each interval.

**closed** : {'left', 'right', 'both', 'neither'}, default 'right'

Whether the intervals are closed on the left-side, right-side, both or neither.

**name** : object, optional

Name to be stored in the index.

**copy** : boolean, default False

copy the data

**dtype** : dtype or None, default None

If None, dtype will be inferred

New in version 0.23.0.

**See also:**

[`interval\_range`](#) Function to create a fixed frequency IntervalIndex

[`IntervalIndex.from\_arrays`](#) Construct an IntervalIndex from a left and right array

[`IntervalIndex.from\_tuples`](#) Construct an IntervalIndex from a list/array of tuples

## Examples

```
>>> pd.IntervalIndex.from_breaks([0, 1, 2, 3])
IntervalIndex([(0, 1], (1, 2], (2, 3])
              closed='right',
              dtype='interval[int64]')
```

### 34.9.1.11 pandas.IntervalIndex.from\_tuples

**classmethod** `IntervalIndex.from_tuples` (*data*, *closed*='right', *name*=None, *copy*=False, *dtype*=None)

Construct an IntervalIndex from a list/array of tuples

**Parameters** **data** : array-like (1-dimensional)

Array of tuples

**closed** : {'left', 'right', 'both', 'neither'}, default 'right'

Whether the intervals are closed on the left-side, right-side, both or neither.

**name** : object, optional

Name to be stored in the index.

**copy** : boolean, default False

by-default copy the data, this is compat only and ignored

**dtype** : dtype or None, default None

If None, dtype will be inferred

New in version 0.23.0.

See also:

[`interval\_range`](#) Function to create a fixed frequency IntervalIndex

[`IntervalIndex.from\_arrays`](#) Construct an IntervalIndex from a left and right array

[`IntervalIndex.from\_breaks`](#) Construct an IntervalIndex from an array of splits

## Examples

```
>>> pd.IntervalIndex.from_tuples([(0, 1), (1, 2)])
IntervalIndex([(0, 1], (1, 2]),
              closed='right', dtype='interval[int64]')
```

### 34.9.1.12 pandas.IntervalIndex.get\_indexer

`IntervalIndex.get_indexer` (*target*, *method*=None, *limit*=None, *tolerance*=None)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to `ndarray.take` to align the current data to the new index.

**Parameters**

**target** [IntervalIndex or list of Intervals]

**method** : {None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

**limit** : int, optional

Maximum number of consecutive labels in `target` to match for inexact matches.

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations most satisfy the equation  $\text{abs}(\text{index}[\text{indexer}] - \text{target}) \leq \text{tolerance}$ .

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

**Returns** **indexer** : ndarray of int

Integers from 0 to `n - 1` indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

## Examples

```
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

### 34.9.1.13 pandas.IntervalIndex.get\_loc

`IntervalIndex.get_loc` (*key*, *method=None*)

Get integer location, slice or boolean mask for requested label.

**Parameters**

**key** [label]

**method** : {None}, optional

- default: matches where the label is within an interval only.

**Returns**

**loc** [int if unique index, slice if monotonic index, else mask]



## Examples

```
>>> i1, i2 = pd.Interval(0, 1), pd.Interval(1, 2)
>>> index = pd.IntervalIndex([i1, i2])
>>> index.get_loc(1)
0
```

You can also supply an interval or an location for a point inside an interval.

```
>>> index.get_loc(pd.Interval(0, 2))
array([0, 1], dtype=int64)
>>> index.get_loc(1.5)
1
```

If a label is in several intervals, you get the locations of all the relevant intervals.

```
>>> i3 = pd.Interval(0, 2)
>>> overlapping_index = pd.IntervalIndex([i2, i3])
>>> overlapping_index.get_loc(1.5)
array([0, 1], dtype=int64)
```

### 34.9.2 IntervalIndex Components

<code>IntervalIndex.from_arrays(left, right[, ...])</code>	Construct from two arrays defining the left and right bounds.
<code>IntervalIndex.from_tuples(data[, closed, ...])</code>	Construct an IntervalIndex from a list/array of tuples
<code>IntervalIndex.from_breaks(breaks[, closed, ...])</code>	Construct an IntervalIndex from an array of splits
<code>IntervalIndex.contains(key)</code>	Return a boolean indicating if the key is IN the index
<code>IntervalIndex.left</code>	Return the left endpoints of each Interval in the IntervalIndex as an Index
<code>IntervalIndex.right</code>	Return the right endpoints of each Interval in the IntervalIndex as an Index
<code>IntervalIndex.mid</code>	Return the midpoint of each Interval in the IntervalIndex as an Index
<code>IntervalIndex.closed</code>	Whether the intervals are closed on the left-side, right-side, both or neither
<code>IntervalIndex.length</code>	Return an Index with entries denoting the length of each Interval in the IntervalIndex
<code>IntervalIndex.values</code>	Return the IntervalIndex's data as a numpy array of Interval objects (with dtype='object')
<code>IntervalIndex.is_non_overlapping_monotonic</code>	Return True if the IntervalIndex is non-overlapping (no Intervals share points) and is either monotonic increasing or monotonic decreasing, else False
<code>IntervalIndex.get_loc(key[, method])</code>	Get integer location, slice or boolean mask for requested label.
<code>IntervalIndex.get_indexer(target[, method, ...])</code>	Compute indexer and mask for new index given the current index.

## 34.10 MultiIndex

---

*MultiIndex*A multi-level, or hierarchical, index object for pandas objects

---

### 34.10.1 pandas.MultiIndex

**class** pandas.**MultiIndex**

A multi-level, or hierarchical, index object for pandas objects

**Parameters** **levels** : sequence of arrays

The unique labels for each level

**labels** : sequence of arrays

Integers for each level designating which label at each location

**sortorder** : optional int

Level of sortedness (must be lexicographically sorted by that level)

**names** : optional sequence of objects

Names for each of the index levels. (name is accepted for compat)

**copy** : boolean, default False

Copy the meta-data

**verify\_integrity** : boolean, default True

Check that the levels/labels are consistent and valid

**See also:***MultiIndex.from\_arrays* Convert list of arrays to MultiIndex*MultiIndex.from\_product* Create a MultiIndex from the cartesian product of iterables*MultiIndex.from\_tuples* Convert list of tuples to a MultiIndex*Index* The base pandas Index type

#### Notes

See the [user guide](#) for more.

#### Examples

A new MultiIndex is typically constructed using one of the helper methods *MultiIndex.from\_arrays()*, *MultiIndex.from\_product()* and *MultiIndex.from\_tuples()*. For example (using *.from\_arrays()*):

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> pd.MultiIndex.from_arrays(arrays, names=('number', 'color'))
MultiIndex(levels=[[1, 2], ['blue', 'red']],
```

(continues on next page)

(continued from previous page)

```
labels=[[0, 0, 1, 1], [1, 0, 1, 0]],
names=['number', 'color'])
```

See further examples for how to construct a MultiIndex in the doc strings of the mentioned helper methods.

## Attributes

<i>names</i>	Names of levels in MultiIndex
<i>nlevels</i>	Integer number of levels in this MultiIndex.
<i>levshape</i>	A tuple with the length of each level.

### 34.10.1.1 pandas.MultiIndex.names

`MultiIndex.names`  
Names of levels in MultiIndex

### 34.10.1.2 pandas.MultiIndex.nlevels

`MultiIndex.nlevels`  
Integer number of levels in this MultiIndex.

### 34.10.1.3 pandas.MultiIndex.levshape

`MultiIndex.levshape`  
A tuple with the length of each level.

<b>levels</b>	
<b>labels</b>	

## Methods

<i>from_arrays</i> (arrays[, sortorder, names])	Convert arrays to MultiIndex
<i>from_tuples</i> (tuples[, sortorder, names])	Convert list of tuples to MultiIndex
<i>from_product</i> (iterables[, sortorder, names])	Make a MultiIndex from the cartesian product of multiple iterables
<i>set_levels</i> (levels[, level, inplace, ...])	Set new levels on MultiIndex.
<i>set_labels</i> (labels[, level, inplace, ...])	Set new labels on MultiIndex.
<i>to_hierarchical</i> (n_repeat[, n_shuffle])	Return a MultiIndex reshaped to conform to the shapes given by <code>n_repeat</code> and <code>n_shuffle</code> .
<i>to_frame</i> ([index])	Create a DataFrame with the levels of the MultiIndex as columns.
<i>is_lexsorted</i> ()	Return True if the labels are lexicographically sorted
<i>sortlevel</i> ([level, ascending, sort_remaining])	Sort MultiIndex at the requested level.
<i>droplevel</i> ([level])	Return Index with requested level removed.
<i>swaplevel</i> ([i, j])	Swap level <code>i</code> with level <code>j</code> .
<i>reorder_levels</i> (order)	Rearrange levels using input order.

Continued on next page

Table 118 – continued from previous page

---

<code>remove_unused_levels()</code>	create a new MultiIndex from the current that removing unused levels, meaning that they are not expressed in the labels
-------------------------------------	---

---

#### 34.10.1.4 `pandas.MultiIndex.from_arrays`

**classmethod** `MultiIndex.from_arrays` (*arrays, sortorder=None, names=None*)

Convert arrays to MultiIndex

**Parameters** `arrays` : list / sequence of array-likes

Each array-like gives one level's value for each data point. `len(arrays)` is the number of levels.

**sortorder** : int or None

Level of sortedness (must be lexicographically sorted by that level)

**Returns**

**index** [MultiIndex]

**See also:**

*`MultiIndex.from_tuples`* Convert list of tuples to MultiIndex

*`MultiIndex.from_product`* Make a MultiIndex from cartesian product of iterables

#### Examples

```
>>> arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
>>> MultiIndex.from_arrays(arrays, names=('number', 'color'))
```

#### 34.10.1.5 `pandas.MultiIndex.from_tuples`

**classmethod** `MultiIndex.from_tuples` (*tuples, sortorder=None, names=None*)

Convert list of tuples to MultiIndex

**Parameters** `tuples` : list / sequence of tuple-likes

Each tuple is the index of one row/column.

**sortorder** : int or None

Level of sortedness (must be lexicographically sorted by that level)

**Returns**

**index** [MultiIndex]

**See also:**

*`MultiIndex.from_arrays`* Convert list of arrays to MultiIndex

*`MultiIndex.from_product`* Make a MultiIndex from cartesian product of iterables

## Examples

```
>>> tuples = [(1, u'red'), (1, u'blue'),
               (2, u'red'), (2, u'blue')]
>>> MultiIndex.from_tuples(tuples, names=('number', 'color'))
```

### 34.10.1.6 pandas.MultiIndex.from\_product

**classmethod** `MultiIndex.from_product` (*iterables, sortorder=None, names=None*)

Make a MultiIndex from the cartesian product of multiple iterables

**Parameters** `iterables` : list / sequence of iterables

Each iterable has unique labels for each level of the index.

`sortorder` : int or None

Level of sortedness (must be lexicographically sorted by that level).

`names` : list / sequence of strings or None

Names for the levels in the index.

**Returns**

`index` [MultiIndex]

See also:

[`MultiIndex.from\_arrays`](#) Convert list of arrays to MultiIndex

[`MultiIndex.from\_tuples`](#) Convert list of tuples to MultiIndex

## Examples

```
>>> numbers = [0, 1, 2]
>>> colors = [u'green', u'purple']
>>> MultiIndex.from_product([numbers, colors],
                             names=['number', 'color'])
MultiIndex(levels=[[0, 1, 2], [u'green', u'purple']],
            labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
            names=[u'number', u'color'])
```

### 34.10.1.7 pandas.MultiIndex.set\_levels

`MultiIndex.set_levels` (*levels, level=None, inplace=False, verify\_integrity=True*)

Set new levels on MultiIndex. Defaults to returning new index.

**Parameters** `levels` : sequence or list of sequence

new level(s) to apply

`level` : int, level name, or sequence of int/level names (default None)

level(s) to set (None for all levels)

`inplace` : bool

if True, mutates in place

**verify\_integrity** : bool (default True)

if True, checks that levels and labels are compatible

#### Returns

**new index (of same type and class...etc)**

### Examples

```
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                (2, u'one'), (2, u'two')],
                                names=['foo', 'bar'])
>>> idx.set_levels(['a', 'b'], [1, 2])
MultiIndex(levels=[[u'a', u'b'], [1, 2]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=[u'foo', u'bar'])
>>> idx.set_levels(['a', 'b'], level=0)
MultiIndex(levels=[[u'a', u'b'], [u'one', u'two']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=[u'foo', u'bar'])
>>> idx.set_levels(['a', 'b'], level='bar')
MultiIndex(levels=[[1, 2], [u'a', u'b']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=[u'foo', u'bar'])
>>> idx.set_levels(['a', 'b'], [1, 2], level=[0, 1])
MultiIndex(levels=[[u'a', u'b'], [1, 2]],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]],
            names=[u'foo', u'bar'])
```

#### 34.10.1.8 pandas.MultiIndex.set\_labels

`MultiIndex.set_labels` (*labels*, *level=None*, *inplace=False*, *verify\_integrity=True*)

Set new labels on MultiIndex. Defaults to returning new index.

**Parameters** **labels** : sequence or list of sequence

new labels to apply

**level** : int, level name, or sequence of int/level names (default None)

level(s) to set (None for all levels)

**inplace** : bool

if True, mutates in place

**verify\_integrity** : bool (default True)

if True, checks that levels and labels are compatible

#### Returns

**new index (of same type and class...etc)**

## Examples

```
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                (2, u'one'), (2, u'two')],
                                names=['foo', 'bar'])
>>> idx.set_labels([[1,0,1,0], [0,0,1,1]])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[1, 0, 1, 0], [0, 0, 1, 1]],
            names=[u'foo', u'bar'])
>>> idx.set_labels([1,0,1,0], level=0)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[1, 0, 1, 0], [0, 1, 0, 1]],
            names=[u'foo', u'bar'])
>>> idx.set_labels([0,0,1,1], level='bar')
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[0, 0, 1, 1], [0, 0, 1, 1]],
            names=[u'foo', u'bar'])
>>> idx.set_labels([[1,0,1,0], [0,0,1,1]], level=[0,1])
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[1, 0, 1, 0], [0, 0, 1, 1]],
            names=[u'foo', u'bar'])
```

### 34.10.1.9 pandas.MultiIndex.to\_hierarchical

`MultiIndex.to_hierarchical(n_repeat, n_shuffle=1)`

Return a MultiIndex reshaped to conform to the shapes given by `n_repeat` and `n_shuffle`.

Useful to replicate and rearrange a MultiIndex for combination with another Index with `n_repeat` items.

**Parameters** `n_repeat` : int

Number of times to repeat the labels on self

`n_shuffle` : int

Controls the reordering of the labels. If the result is going to be an inner level in a MultiIndex, `n_shuffle` will need to be greater than one. The size of each label must be divisible by `n_shuffle`.

**Returns**

**MultiIndex**

## Examples

```
>>> idx = MultiIndex.from_tuples([(1, u'one'), (1, u'two'),
                                (2, u'one'), (2, u'two')])
>>> idx.to_hierarchical(3)
MultiIndex(levels=[[1, 2], [u'one', u'two']],
            labels=[[0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
                   [0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1]])
```

#### 34.10.1.10 pandas.MultiIndex.to\_frame

`MultiIndex.to_frame(index=True)`

Create a DataFrame with the levels of the MultiIndex as columns.

New in version 0.20.0.

**Parameters** `index` : boolean, default True

Set the index of the returned DataFrame as the original MultiIndex.

**Returns**

**DataFrame** [a DataFrame containing the original MultiIndex data.]

#### 34.10.1.11 pandas.MultiIndex.is\_lexsorted

`MultiIndex.is_lexsorted()`

Return True if the labels are lexicographically sorted

#### 34.10.1.12 pandas.MultiIndex.sortlevel

`MultiIndex.sortlevel(level=0, ascending=True, sort_remaining=True)`

Sort MultiIndex at the requested level. The result will respect the original ordering of the associated factor at that level.

**Parameters** `level` : list-like, int or str, default 0

If a string is given, must be a name of the level If list-like must be names or ints of levels.

**ascending** : boolean, default True

False to sort in descending order Can also be a list to specify a directed ordering

**sort\_remaining** [sort by the remaining levels after level.]

**Returns** `sorted_index` : pd.MultiIndex

Resulting index

**indexer** : np.ndarray

Indices of output values in original index

#### 34.10.1.13 pandas.MultiIndex.droplevel

`MultiIndex.droplevel(level=0)`

Return Index with requested level removed. If MultiIndex has only 2 levels, the result will be of Index type not MultiIndex.

**Parameters**

**level** [int/level name or list thereof]

**Returns**

**index** [Index or MultiIndex]



## Notes

Does not check if result index is unique or not

### 34.10.1.14 pandas.MultiIndex.swaplevel

`MultiIndex.swaplevel` (*i*=-2, *j*=-1)

Swap level *i* with level *j*.

Calling this method does not change the ordering of the values.

**Parameters** *i* : int, str, default -2

First level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

*j* : int, str, default -1

Second level of index to be swapped. Can pass level name as string. Type of parameters can be mixed.

**Returns** `MultiIndex`

A new `MultiIndex`

**.. versionchanged:: 0.18.1**

The indexes *i* and *j* are now optional, and default to the two innermost levels of the index.

**See also:**

[`Series.swaplevel`](#) Swap levels *i* and *j* in a `MultiIndex`

[`Dataframe.swaplevel`](#) Swap levels *i* and *j* in a `MultiIndex` on a particular axis

## Examples

```

>>> mi = pd.MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
...                     labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> mi
MultiIndex(levels=[['a', 'b'], ['bb', 'aa']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
>>> mi.swaplevel(0, 1)
MultiIndex(levels=[['bb', 'aa'], ['a', 'b']],
            labels=[[0, 1, 0, 1], [0, 0, 1, 1]])

```

### 34.10.1.15 pandas.MultiIndex.reorder\_levels

`MultiIndex.reorder_levels` (*order*)

Rearrange levels using input order. May not drop or duplicate levels

### 34.10.1.16 pandas.MultiIndex.remove\_unused\_levels

`MultiIndex.remove_unused_levels()`

create a new MultiIndex from the current that removing unused levels, meaning that they are not expressed in the labels

The resulting MultiIndex will have the same outward appearance, meaning the same `.values` and ordering. It will also be `.equals()` to the original.

New in version 0.20.0.

#### Returns

**MultiIndex**

#### Examples

```
>>> i = pd.MultiIndex.from_product([range(2), list('ab')])
MultiIndex(levels=[[0, 1], ['a', 'b']],
            labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

```
>>> i[2:]
MultiIndex(levels=[[0, 1], ['a', 'b']],
            labels=[[1, 1], [0, 1]])
```

The 0 from the first level is not represented and can be removed

```
>>> i[2:].remove_unused_levels()
MultiIndex(levels=[[1], ['a', 'b']],
            labels=[[0, 0], [0, 1]])
```

---

*IndexSlice*

Create an object to more easily perform multi-index slicing

---

## 34.10.2 pandas.IndexSlice

`pandas.IndexSlice = <pandas.core.indexing._IndexSlice object>`

Create an object to more easily perform multi-index slicing

#### Examples

```
>>> midx = pd.MultiIndex.from_product([['A0', 'A1'], ['B0', 'B1', 'B2', 'B3']])
>>> columns = ['foo', 'bar']
>>> dfmi = pd.DataFrame(np.arange(16).reshape((len(midx), len(columns))),
                        index=midx, columns=columns)
```

Using the default slice command:

```
>>> dfmi.loc[(slice(None), slice('B0', 'B1')), :]
      foo  bar
A0 B0    0    1
   B1    2    3
```

(continues on next page)

(continued from previous page)

A1	B0	8	9
	B1	10	11

Using the `IndexSlice` class for a more intuitive command:

```
>>> idx = pd.IndexSlice
>>> dfmi.loc[idx[:, 'B0':'B1'], :]
      foo  bar
A0 B0    0    1
   B1    2    3
A1 B0    8    9
   B1   10   11
```

### 34.10.3 MultiIndex Constructors

<code>MultiIndex.from_arrays(arrays[, ...])</code>	<code>sortorder</code>	Convert arrays to MultiIndex
<code>MultiIndex.from_tuples(tuples[, ...])</code>	<code>sortorder</code>	Convert list of tuples to MultiIndex
<code>MultiIndex.from_product(iterables[, ...])</code>		Make a MultiIndex from the cartesian product of multiple iterables

### 34.10.4 MultiIndex Attributes

<code>MultiIndex.names</code>	Names of levels in MultiIndex
<code>MultiIndex.levels</code>	
<code>MultiIndex.labels</code>	
<code>MultiIndex.nlevels</code>	Integer number of levels in this MultiIndex.
<code>MultiIndex.levshape</code>	A tuple with the length of each level.

#### 34.10.4.1 pandas.MultiIndex.levels

`MultiIndex.levels`

#### 34.10.4.2 pandas.MultiIndex.labels

`MultiIndex.labels`

### 34.10.5 MultiIndex Components

<code>MultiIndex.set_levels(levels[, level, ...])</code>	Set new levels on MultiIndex.
<code>MultiIndex.set_labels(labels[, level, ...])</code>	Set new labels on MultiIndex.
<code>MultiIndex.to_hierarchical(n_repeat[, n_shuffle])</code>	Return a MultiIndex reshaped to conform to the shapes given by <code>n_repeat</code> and <code>n_shuffle</code> .
<code>MultiIndex.to_frame([index])</code>	Create a DataFrame with the levels of the MultiIndex as columns.

Continued on next page

Table 122 – continued from previous page

<code>MultiIndex.is_lexsorted()</code>	Return True if the labels are lexicographically sorted
<code>MultiIndex.sortlevel([level, ascending, ...])</code>	Sort MultiIndex at the requested level.
<code>MultiIndex.droplevel([level])</code>	Return Index with requested level removed.
<code>MultiIndex.swaplevel([i, j])</code>	Swap level i with level j.
<code>MultiIndex.reorder_levels(order)</code>	Rearrange levels using input order.
<code>MultiIndex.remove_unused_levels()</code>	create a new MultiIndex from the current that removing unused levels, meaning that they are not expressed in the labels
<code>MultiIndex.unique([level])</code>	Return unique values in the index.

### 34.10.5.1 pandas.MultiIndex.unique

`MultiIndex.unique` (*level=None*)

Return unique values in the index. Uniques are returned in order of appearance, this does NOT sort.

**Parameters** `level` : int or str, optional, default None

Only return values from specified level (for MultiIndex)

New in version 0.23.0.

**Returns**

Index without duplicates

**See also:**

`unique`, `Series.unique`

## 34.10.6 MultiIndex Selecting

<code>MultiIndex.get_loc(key[, method])</code>	Get location for a label or a tuple of labels as an integer, slice or boolean mask.
<code>MultiIndex.get_indexer(target[, method, ...])</code>	Compute indexer and mask for new index given the current index.
<code>MultiIndex.get_level_values(level)</code>	Return vector of label values for requested level, equal to the length of the index.

### 34.10.6.1 pandas.MultiIndex.get\_loc

`MultiIndex.get_loc` (*key, method=None*)

Get location for a label or a tuple of labels as an integer, slice or boolean mask.

**Parameters**

**key** [label or tuple of labels (one for each level)]

**method** [None]

**Returns** `loc` : int, slice object or boolean mask

If the key is past the lexsort depth, the return may be a boolean mask array, otherwise it is always a slice or int.

**See also:**

`Index.get_loc` `get_loc` method for (single-level) index.

**MultiIndex.slice\_locs** Get slice location given start label(s) and end label(s).

**MultiIndex.get\_locs** Get location for a label/slice/list/mask or a sequence of such.

## Notes

The key cannot be a slice, list of same-level labels, a boolean mask, or a sequence of such. If you want to use those, use `MultiIndex.get_locs()` instead.

## Examples

```
>>> mi = pd.MultiIndex.from_arrays([list('abb'), list('def')])
```

```
>>> mi.get_loc('b')
slice(1, 3, None)
```

```
>>> mi.get_loc(('b', 'e'))
1
```

### 34.10.6.2 pandas.MultiIndex.get\_indexer

**MultiIndex.get\_indexer** (*target*, *method=None*, *limit=None*, *tolerance=None*)

Compute indexer and mask for new index given the current index. The indexer should be then used as an input to `ndarray.take` to align the current data to the new index.

#### Parameters

**target** [MultiIndex or list of tuples]

**method** : {None, 'pad'/'ffill', 'backfill'/'bfill', 'nearest'}, optional

- default: exact matches only.
- pad / ffill: find the PREVIOUS index value if no exact match.
- backfill / bfill: use NEXT index value if no exact match
- nearest: use the NEAREST index value if no exact match. Tied distances are broken by preferring the larger index value.

**limit** : int, optional

Maximum number of consecutive labels in *target* to match for inexact matches.

**tolerance** : optional

Maximum distance between original and new labels for inexact matches. The values of the index at the matching locations must satisfy the equation `abs(index[indexer] - target) <= tolerance`.

Tolerance may be a scalar value, which applies the same tolerance to all values, or list-like, which applies variable tolerance per element. List-like includes list, tuple, array, Series, and must be the same size as the index and its dtype must exactly match the index's type.

New in version 0.21.0: (list-like tolerance)

**Returns** **indexer** : ndarray of int

Integers from 0 to  $n - 1$  indicating that the index at these positions matches the corresponding target values. Missing values in the target are marked by -1.

### Examples

```
>>> indexer = index.get_indexer(new_index)
>>> new_values = cur_values.take(indexer)
```

#### 34.10.6.3 pandas.MultiIndex.get\_level\_values

`MultiIndex.get_level_values` (*level*)

Return vector of label values for requested level, equal to the length of the index.

**Parameters** *level* : int or str

*level* is either the integer position of the level in the MultiIndex, or the name of the level.

**Returns** *values* : Index

*values* is a level of this MultiIndex converted to a single *Index* (or subclass thereof).

### Examples

Create a MultiIndex:

```
>>> mi = pd.MultiIndex.from_arrays((list('abc'), list('def')))
>>> mi.names = ['level_1', 'level_2']
```

Get level values by supplying level as either integer or name:

```
>>> mi.get_level_values(0)
Index(['a', 'b', 'c'], dtype='object', name='level_1')
>>> mi.get_level_values('level_2')
Index(['d', 'e', 'f'], dtype='object', name='level_2')
```

## 34.11 DatetimeIndex

---

*DatetimeIndex*

Immutable ndarray of datetime64 data, represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry meta-data such as frequency information.

---

### 34.11.1 pandas.DatetimeIndex

**class** pandas.DatetimeIndex

Immutable ndarray of datetime64 data, represented internally as int64, and which can be boxed to Timestamp objects that are subclasses of datetime and carry metadata such as frequency information.

**Parameters** *data* : array-like (1-dimensional), optional

Optional datetime-like data to construct index with

**copy** : bool

Make a copy of input ndarray

**freq** : string or pandas offset object, optional

One of pandas date offset strings or corresponding objects

**start** : starting value, datetime-like, optional

If data is None, start is used as the start point in generating regular timestamp data.

**periods** : int, optional, > 0

Number of periods to generate, if generating index. Takes precedence over end argument

**end** : end time, datetime-like, optional

If periods is none, generated index will extend to first conforming time on or just past end argument

**closed** : string or None, default None

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

**tz** [pytz.timezone or dateutil.tz.tzfile]

**ambiguous** : 'infer', bool-ndarray, 'NaT', default 'raise'

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an AmbiguousTimeError if there are ambiguous times

**name** : object

Name to be stored in the index

**dayfirst** : bool, default False

If True, parse dates in *data* with the day first order

**yearfirst** : bool, default False

If True parse dates in *data* with the year first order

See also:

**Index** The base pandas Index type

**TimedeltaIndex** Index of timedelta64 data

**PeriodIndex** Index of Period data

**pandas.to\_datetime** Convert argument to datetime

## Notes

To learn more about the frequency strings, please see [this link](#).

## Attributes

<i>year</i>	The year of the datetime
<i>month</i>	The month as January=1, December=12
<i>day</i>	The days of the datetime
<i>hour</i>	The hours of the datetime
<i>minute</i>	The minutes of the datetime
<i>second</i>	The seconds of the datetime
<i>microsecond</i>	The microseconds of the datetime
<i>nanosecond</i>	The nanoseconds of the datetime
<i>date</i>	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without time-zone information).
<i>time</i>	Returns numpy array of datetime.time.
<i>dayofyear</i>	The ordinal day of the year
<i>weekofyear</i>	The week ordinal of the year
<i>week</i>	The week ordinal of the year
<i>dayofweek</i>	The day of the week with Monday=0, Sunday=6
<i>weekday</i>	The day of the week with Monday=0, Sunday=6
<i>quarter</i>	The quarter of the date
<i>freq</i>	Return the frequency object if it is set, otherwise None
<i>freqstr</i>	Return the frequency object as a string if it is set, otherwise None
<i>is_month_start</i>	Logical indicating if first day of month (defined by frequency)
<i>is_month_end</i>	Indicator for whether the date is the last day of the month.
<i>is_quarter_start</i>	Indicator for whether the date is the first day of a quarter.
<i>is_quarter_end</i>	Indicator for whether the date is the last day of a quarter.
<i>is_year_start</i>	Indicate whether the date is the first day of a year.
<i>is_year_end</i>	Indicate whether the date is the last day of the year.
<i>is_leap_year</i>	Boolean indicator if the date belongs to a leap year.
<i>inferred_freq</i>	Tries to return a string representing a frequency guess, generated by infer_freq.

### 34.11.1.1 pandas.DatetimeIndex.year

DatetimeIndex.**year**

The year of the datetime



#### 34.11.1.2 pandas.DatetimeIndex.month

`DatetimeIndex.month`

The month as January=1, December=12

#### 34.11.1.3 pandas.DatetimeIndex.day

`DatetimeIndex.day`

The days of the datetime

#### 34.11.1.4 pandas.DatetimeIndex.hour

`DatetimeIndex.hour`

The hours of the datetime

#### 34.11.1.5 pandas.DatetimeIndex.minute

`DatetimeIndex.minute`

The minutes of the datetime

#### 34.11.1.6 pandas.DatetimeIndex.second

`DatetimeIndex.second`

The seconds of the datetime

#### 34.11.1.7 pandas.DatetimeIndex.microsecond

`DatetimeIndex.microsecond`

The microseconds of the datetime

#### 34.11.1.8 pandas.DatetimeIndex.nanosecond

`DatetimeIndex.nanosecond`

The nanoseconds of the datetime

#### 34.11.1.9 pandas.DatetimeIndex.date

`DatetimeIndex.date`

Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without time-zone information).

#### 34.11.1.10 pandas.DatetimeIndex.time

`DatetimeIndex.time`

Returns numpy array of datetime.time. The time part of the Timestamps.

#### 34.11.1.11 pandas.DatetimeIndex.dayofyear

`DatetimeIndex.dayofyear`  
The ordinal day of the year

#### 34.11.1.12 pandas.DatetimeIndex.weekofyear

`DatetimeIndex.weekofyear`  
The week ordinal of the year

#### 34.11.1.13 pandas.DatetimeIndex.week

`DatetimeIndex.week`  
The week ordinal of the year

#### 34.11.1.14 pandas.DatetimeIndex.dayofweek

`DatetimeIndex.dayofweek`  
The day of the week with Monday=0, Sunday=6

#### 34.11.1.15 pandas.DatetimeIndex.weekday

`DatetimeIndex.weekday`  
The day of the week with Monday=0, Sunday=6

#### 34.11.1.16 pandas.DatetimeIndex.quarter

`DatetimeIndex.quarter`  
The quarter of the date

#### 34.11.1.17 pandas.DatetimeIndex.freq

`DatetimeIndex.freq`  
Return the frequency object if it is set, otherwise None

#### 34.11.1.18 pandas.DatetimeIndex.freqstr

`DatetimeIndex.freqstr`  
Return the frequency object as a string if it is set, otherwise None

#### 34.11.1.19 pandas.DatetimeIndex.is\_month\_start

`DatetimeIndex.is_month_start`  
Logical indicating if first day of month (defined by frequency)

### 34.11.1.20 pandas.DatetimeIndex.is\_month\_end

`DatetimeIndex.is_month_end`

Indicator for whether the date is the last day of the month.

**Returns Series or array**

For Series, returns a Series with boolean values. For DatetimeIndex, returns a boolean array.

**See also:**

[`is\_month\_start`](#) Indicator for whether the date is the first day of the month.

#### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```

>>> dates = pd.Series(pd.date_range("2018-02-27", periods=3))
>>> dates
0    2018-02-27
1    2018-02-28
2    2018-03-01
dtype: datetime64[ns]
>>> dates.dt.is_month_end
0    False
1     True
2    False
dtype: bool

```

```

>>> idx = pd.date_range("2018-02-27", periods=3)
>>> idx.is_month_end
array([False,  True, False], dtype=bool)

```

### 34.11.1.21 pandas.DatetimeIndex.is\_quarter\_start

`DatetimeIndex.is_quarter_start`

Indicator for whether the date is the first day of a quarter.

**Returns `is_quarter_start` : Series or DatetimeIndex**

The same type as the original data with boolean values. Series will have the same name and index. `DatetimeIndex` will have the same name.

**See also:**

[`quarter`](#) Return the quarter of the date.

[`is\_quarter\_end`](#) Similar property for indicating the quarter start.

#### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                                           periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_start=df.dates.dt.is_quarter_start)
   dates      quarter  is_quarter_start
0 2017-03-30         1             False
1 2017-03-31         1             False
2 2017-04-01         2              True
3 2017-04-02         2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_start
array([False, False,  True, False])
```

### 34.11.1.22 pandas.DatetimeIndex.is\_quarter\_end

**DatetimeIndex.is\_quarter\_end**

Indicator for whether the date is the last day of a quarter.

**Returns is\_quarter\_end** : Series or DatetimeIndex

The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

**See also:**

**quarter** Return the quarter of the date.

**is\_quarter\_start** Similar property indicating the quarter start.

### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> df = pd.DataFrame({'dates': pd.date_range("2017-03-30",
...                                           periods=4)})
>>> df.assign(quarter=df.dates.dt.quarter,
...           is_quarter_end=df.dates.dt.is_quarter_end)
   dates      quarter  is_quarter_end
0 2017-03-30         1             False
1 2017-03-31         1              True
2 2017-04-01         2             False
3 2017-04-02         2             False
```

```
>>> idx = pd.date_range('2017-03-30', periods=4)
>>> idx
DatetimeIndex(['2017-03-30', '2017-03-31', '2017-04-01', '2017-04-02'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_quarter_end
array([False,  True, False, False])
```

### 34.11.1.23 pandas.DatetimeIndex.is\_year\_start

`DatetimeIndex.is_year_start`

Indicate whether the date is the first day of a year.

#### Returns Series or DatetimeIndex

The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

[`is\_year\_end`](#) Similar property indicating the last day of the year.

### Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0    2017-12-30
1    2017-12-31
2    2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_start
0    False
1    False
2     True
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_start
array([False, False,  True])
```

### 34.11.1.24 pandas.DatetimeIndex.is\_year\_end

`DatetimeIndex.is_year_end`

Indicate whether the date is the last day of the year.

#### Returns Series or DatetimeIndex

The same type as the original data with boolean values. Series will have the same name and index. DatetimeIndex will have the same name.

See also:

*is\_year\_start* Similar property indicating the start of the year.

## Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> dates = pd.Series(pd.date_range("2017-12-30", periods=3))
>>> dates
0    2017-12-30
1    2017-12-31
2    2018-01-01
dtype: datetime64[ns]
```

```
>>> dates.dt.is_year_end
0    False
1     True
2    False
dtype: bool
```

```
>>> idx = pd.date_range("2017-12-30", periods=3)
>>> idx
DatetimeIndex(['2017-12-30', '2017-12-31', '2018-01-01'],
              dtype='datetime64[ns]', freq='D')
```

```
>>> idx.is_year_end
array([False,  True, False])
```

### 34.11.1.25 pandas.DatetimeIndex.is\_leap\_year

`DatetimeIndex.is_leap_year`

Boolean indicator if the date belongs to a leap year.

A leap year is a year, which has 366 days (instead of 365) including 29th of February as an intercalary day. Leap years are years which are multiples of four with the exception of years divisible by 100 but not by 400.

#### Returns Series or ndarray

Booleans indicating if dates belong to a leap year.

## Examples

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

```
>>> idx = pd.date_range("2012-01-01", "2015-01-01", freq="Y")
>>> idx
DatetimeIndex(['2012-12-31', '2013-12-31', '2014-12-31'],
              dtype='datetime64[ns]', freq='A-DEC')
>>> idx.is_leap_year
array([ True, False, False], dtype=bool)
```

```

>>> dates = pd.Series(idx)
>>> dates_series
0    2012-12-31
1    2013-12-31
2    2014-12-31
dtype: datetime64[ns]
>>> dates_series.dt.is_leap_year
0     True
1    False
2    False
dtype: bool

```

### 34.11.1.26 pandas.DatetimeIndex.inferred\_freq

#### DatetimeIndex.inferred\_freq

Tries to return a string representing a frequency guess, generated by `infer_freq`. Returns `None` if it can't autodetect the frequency.

tz	
----	--

#### Methods

<code>normalize()</code>	Convert times to midnight.
<code>strftime(date_format)</code>	Convert to Index using specified <code>date_format</code> .
<code>snap([freq])</code>	Snap time stamps to nearest occurring frequency
<code>tz_convert(tz)</code>	Convert tz-aware DatetimeIndex from one time zone to another.
<code>tz_localize(tz[, ambiguous, errors])</code>	Localize tz-naive DatetimeIndex to tz-aware DatetimeIndex.
<code>round(freq, *args, **kwargs)</code>	round the data to the specified <code>freq</code> .
<code>floor(freq)</code>	floor the data to the specified <code>freq</code> .
<code>ceil(freq)</code>	ceil the data to the specified <code>freq</code> .
<code>to_period([freq])</code>	Cast to PeriodIndex at a particular frequency.
<code>to_perioddelta(freq)</code>	Calculate TimedeltaIndex of difference between index values and index converted to periodIndex at specified freq.
<code>to_pydatetime()</code>	Return DatetimeIndex as object ndarray of date-time.datetime objects
<code>to_series([keep_tz, index, name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
<code>to_frame([index])</code>	Create a DataFrame with a column containing the Index.
<code>month_name([locale])</code>	Return the month names of the DateTimeIndex with specified locale.
<code>day_name([locale])</code>	Return the day names of the DateTimeIndex with specified locale.

### 34.11.1.27 pandas.DatetimeIndex.normalize

`DatetimeIndex.normalize()`

Convert times to midnight.

The time component of the date-time is converted to midnight i.e. 00:00:00. This is useful in cases, when the time does not matter. Length is unaltered. The timezones are unaffected.

This method is available on Series with datetime values under the `.dt` accessor, and directly on `DatetimeIndex`.

#### Returns `DatetimeIndex` or `Series`

The same type as the original data. Series will have the same name and index. `DatetimeIndex` will have the same name.

See also:

[`floor`](#) Floor the datetimes to the specified freq.

[`ceil`](#) Ceil the datetimes to the specified freq.

[`round`](#) Round the datetimes to the specified freq.

#### Examples

```
>>> idx = pd.DatetimeIndex(start='2014-08-01 10:00', freq='H',
...                        periods=3, tz='Asia/Calcutta')
>>> idx
DatetimeIndex(['2014-08-01 10:00:00+05:30',
               '2014-08-01 11:00:00+05:30',
               '2014-08-01 12:00:00+05:30'],
              dtype='datetime64[ns, Asia/Calcutta]', freq='H')
>>> idx.normalize()
DatetimeIndex(['2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30',
               '2014-08-01 00:00:00+05:30'],
              dtype='datetime64[ns, Asia/Calcutta]', freq=None)
```

### 34.11.1.28 pandas.DatetimeIndex.strftime

`DatetimeIndex.strftime(date_format)`

Convert to Index using specified `date_format`.

Return an Index of formatted strings specified by `date_format`, which supports the same string format as the python standard library. Details of the string format can be found in [python string format doc](#)

**Parameters** `date_format`: str

Date format string (e.g. “%Y-%m-%d”).

**Returns** `Index`

Index of formatted strings

See also:

[`pandas.to\_datetime`](#) Convert the given argument to datetime



***DatetimeIndex.normalize*** Return DatetimeIndex with times to midnight.

***DatetimeIndex.round*** Round the DatetimeIndex to the specified freq.

***DatetimeIndex.floor*** Floor the DatetimeIndex to the specified freq.

## Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                      periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

### 34.11.1.29 pandas.DatetimeIndex.snap

**DatetimeIndex.snap** (*freq='S'*)

Snap time stamps to nearest occurring frequency

### 34.11.1.30 pandas.DatetimeIndex.tz\_convert

**DatetimeIndex.tz\_convert** (*tz*)

Convert tz-aware DatetimeIndex from one time zone to another.

**Parameters** *tz* : string, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time. Corresponding timestamps would be converted to this time zone of the DatetimeIndex. A *tz* of None will convert to UTC and remove the timezone information.

**Returns**

**normalized** [DatetimeIndex]

**Raises** **TypeError**

If DatetimeIndex is tz-naive.

**See also:**

***DatetimeIndex.tz*** A timezone that has a variable offset from UTC

***DatetimeIndex.tz\_localize*** Localize tz-naive DatetimeIndex to a given time zone, or remove timezone from a tz-aware DatetimeIndex.

## Examples

With the *tz* parameter, we can change the DatetimeIndex to other time zones:

```
>>> dti = pd.DatetimeIndex(start='2014-08-01 09:00',
...                          freq='H', periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert('US/Central')
DatetimeIndex(['2014-08-01 02:00:00-05:00',
               '2014-08-01 03:00:00-05:00',
               '2014-08-01 04:00:00-05:00'],
              dtype='datetime64[ns, US/Central]', freq='H')
```

With the `tz=None`, we can remove the timezone (after converting to UTC if necessary):

```
>>> dti = pd.DatetimeIndex(start='2014-08-01 09:00', freq='H',
...                          periods=3, tz='Europe/Berlin')
```

```
>>> dti
DatetimeIndex(['2014-08-01 09:00:00+02:00',
               '2014-08-01 10:00:00+02:00',
               '2014-08-01 11:00:00+02:00'],
              dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

```
>>> dti.tz_convert(None)
DatetimeIndex(['2014-08-01 07:00:00',
               '2014-08-01 08:00:00',
               '2014-08-01 09:00:00'],
              dtype='datetime64[ns]', freq='H')
```

### 34.11.1.31 pandas.DatetimeIndex.tz\_localize

`DatetimeIndex.tz_localize(tz, ambiguous='raise', errors='raise')`

Localize tz-naive `DatetimeIndex` to tz-aware `DatetimeIndex`.

This method takes a time zone (`tz`) naive `DatetimeIndex` object and makes this time zone aware. It does not move the time to another time zone. Time zone localization helps to switch from time zone aware to time zone unaware objects.

**Parameters** `tz` : string, `pytz.timezone`, `dateutil.tz.tzfile` or `None`

Time zone to convert timestamps to. Passing `None` will remove the time zone information preserving local time.

**ambiguous** : str { 'infer', 'NaT', 'raise' } or bool array, default 'raise'

- 'infer' will attempt to infer fall dst-transition hours based on order
- bool-ndarray where True signifies a DST time, False signifies a non-DST time (note that this flag is only applicable for ambiguous times)
- 'NaT' will return NaT where there are ambiguous times
- 'raise' will raise an `AmbiguousTimeError` if there are ambiguous times

**errors** : { 'raise', 'coerce' }, default 'raise'

- 'raise' will raise a `NonExistentTimeError` if a timestamp is not valid in the specified time zone (e.g. due to a transition from or to DST time)

- ‘coerce’ will return NaT if the timestamp can not be converted to the specified time zone

New in version 0.19.0.

#### Returns DatetimeIndex

Index converted to the specified time zone.

#### Raises TypeError

If the DatetimeIndex is tz-aware and tz is not None.

See also:

[`DatetimeIndex.tz\_convert`](#) Convert tz-aware DatetimeIndex from one time zone to another.

### Examples

```
>>> tz_naive = pd.date_range('2018-03-01 09:00', periods=3)
>>> tz_naive
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
              '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

Localize DatetimeIndex in US/Eastern time zone:

```
>>> tz_aware = tz_naive.tz_localize(tz='US/Eastern')
>>> tz_aware
DatetimeIndex(['2018-03-01 09:00:00-05:00',
              '2018-03-02 09:00:00-05:00',
              '2018-03-03 09:00:00-05:00'],
              dtype='datetime64[ns, US/Eastern]', freq='D')
```

With the tz=None, we can remove the time zone information while keeping the local time (not converted to UTC):

```
>>> tz_aware.tz_localize(None)
DatetimeIndex(['2018-03-01 09:00:00', '2018-03-02 09:00:00',
              '2018-03-03 09:00:00'],
              dtype='datetime64[ns]', freq='D')
```

#### 34.11.1.32 pandas.DatetimeIndex.round

`DatetimeIndex.round(freq, *args, **kwargs)`  
round the data to the specified *freq*.

**Parameters** *freq*: str or Offset

The frequency level to round the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** DatetimeIndex, TimedeltaIndex, or Series

Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises**

**ValueError if the ‘freq’ cannot be converted.**

## Examples

### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

### Series

```
>>> pd.Series(rng).dt.round("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

### 34.11.1.33 pandas.DatetimeIndex.floor

`DatetimeIndex.floor(freq)`

floor the data to the specified *freq*.

**Parameters** *freq* : str or Offset

The frequency level to floor the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** `DatetimeIndex`, `TimedeltaIndex`, or `Series`

Index of the same type for a `DatetimeIndex` or `TimedeltaIndex`, or a `Series` with the same index for a `Series`.

**Raises**

**ValueError** if the ‘freq’ cannot be converted.

## Examples

### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq='H')
```

(continues on next page)

(continued from previous page)

```
'2018-01-01 12:00:00'],
dtype='datetime64[ns]', freq=None)
```

**Series**

```
>>> pd.Series(rng).dt.floor("H")
0    2018-01-01 11:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

**34.11.1.34 pandas.DatetimeIndex.ceil****DatetimeIndex.ceil** (*freq*)ceil the data to the specified *freq*.**Parameters** *freq* : str or Offset

The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** **DatetimeIndex, TimedeltaIndex, or Series**

Index of the same type for a DatetimeIndex or TimedeltaIndex, or a Series with the same index for a Series.

**Raises****ValueError** if the ‘freq’ cannot be converted.**Examples****DatetimeIndex**

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
              '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
              '2018-01-01 13:00:00'],
              dtype='datetime64[ns]', freq=None)
```

**Series**

```
>>> pd.Series(rng).dt.ceil("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 13:00:00
dtype: datetime64[ns]
```

### 34.11.1.35 pandas.DatetimeIndex.to\_period

`DatetimeIndex.to_period` (*freq=None*)

Cast to PeriodIndex at a particular frequency.

Converts DatetimeIndex to PeriodIndex.

**Parameters** `freq` : string or Offset, optional

One of pandas' *offset strings* or an Offset object. Will be inferred by default.

**Returns**

**PeriodIndex**

**Raises** `ValueError`

When converting a DatetimeIndex with non-regular values, so that a frequency cannot be inferred.

**See also:**

*pandas.PeriodIndex* Immutable ndarray holding ordinal values

*pandas.DatetimeIndex.to\_pydatetime* Return DatetimeIndex as object

#### Examples

```
>>> df = pd.DataFrame({"y": [1,2,3]},
...                    index=pd.to_datetime(["2000-03-31 00:00:00",
...                    "2000-05-31 00:00:00",
...                    "2000-08-31 00:00:00"]))
>>> df.index.to_period("M")
PeriodIndex(['2000-03', '2000-05', '2000-08'],
            dtype='period[M]', freq='M')
```

Infer the daily frequency

```
>>> idx = pd.date_range("2017-01-01", periods=2)
>>> idx.to_period()
PeriodIndex(['2017-01-01', '2017-01-02'],
            dtype='period[D]', freq='D')
```

### 34.11.1.36 pandas.DatetimeIndex.to\_perioddelta

`DatetimeIndex.to_perioddelta` (*freq*)

Calculate TimedeltaIndex of difference between index values and index converted to periodIndex at specified freq. Used for vectorized offsets

**Parameters**

**freq**: Period frequency

**Returns**

**y**: TimedeltaIndex

### 34.11.1.37 pandas.DatetimeIndex.to\_pydatetime

`DatetimeIndex.to_pydatetime()`

Return `DatetimeIndex` as object ndarray of `datetime.datetime` objects

#### Returns

**datetimes** [ndarray]

### 34.11.1.38 pandas.DatetimeIndex.to\_series

`DatetimeIndex.to_series(keep_tz=False, index=None, name=None)`

Create a `Series` with both index and values equal to the index keys useful with `map` for returning an indexer based on an index

**Parameters** `keep_tz` : optional, defaults `False`.

return the data keeping the timezone.

If `keep_tz` is `True`:

If the timezone is not set, the resulting `Series` will have a `datetime64[ns]` dtype.

Otherwise the `Series` will have an `datetime64[ns, tz]` dtype; the `tz` will be preserved.

If `keep_tz` is `False`:

`Series` will have a `datetime64[ns]` dtype. TZ aware objects will have the `tz` removed.

**index** : Index, optional

index of resulting `Series`. If `None`, defaults to original index

**name** : string, optional

name of resulting `Series`. If `None`, defaults to name of original index

#### Returns

**Series**

### 34.11.1.39 pandas.DatetimeIndex.to\_frame

`DatetimeIndex.to_frame(index=True)`

Create a `DataFrame` with a column containing the Index.

New in version 0.21.0.

**Parameters** `index` : boolean, default `True`

Set the index of the returned `DataFrame` as the original Index.

#### Returns DataFrame

`DataFrame` containing the original Index data.

**See also:**

[`Index.to\_series`](#) Convert an Index to a Series.

[`Series.to\_frame`](#) Convert Series to DataFrame.

## Examples

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
      animal
animal
Ant      Ant
Bear    Bear
Cow     Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
      animal
0      Ant
1     Bear
2      Cow
```

### 34.11.1.40 pandas.DatetimeIndex.month\_name

`DatetimeIndex.month_name(locale=None)`

Return the month names of the DateTimeIndex with specified locale.

**Parameters** `locale` : string, default None (English locale)

locale determining the language in which to return the month name

**Returns** `month_names` : Index

Index of month names

.. versionadded:: 0.23.0

### 34.11.1.41 pandas.DatetimeIndex.day\_name

`DatetimeIndex.day_name(locale=None)`

Return the day names of the DateTimeIndex with specified locale.

**Parameters** `locale` : string, default None (English locale)

locale determining the language in which to return the day name

**Returns** `month_names` : Index

Index of day names

.. versionadded:: 0.23.0

## 34.11.2 Time/Date Components

<code>DatetimeIndex.year</code>	The year of the datetime
<code>DatetimeIndex.month</code>	The month as January=1, December=12
<code>DatetimeIndex.day</code>	The days of the datetime
<code>DatetimeIndex.hour</code>	The hours of the datetime

Continued on next page



Table 127 – continued from previous page

<code>DatetimeIndex.minute</code>	The minutes of the datetime
<code>DatetimeIndex.second</code>	The seconds of the datetime
<code>DatetimeIndex.microsecond</code>	The microseconds of the datetime
<code>DatetimeIndex.nanosecond</code>	The nanoseconds of the datetime
<code>DatetimeIndex.date</code>	Returns numpy array of python datetime.date objects (namely, the date part of Timestamps without timezone information).
<code>DatetimeIndex.time</code>	Returns numpy array of datetime.time.
<code>DatetimeIndex.dayofyear</code>	The ordinal day of the year
<code>DatetimeIndex.weekofyear</code>	The week ordinal of the year
<code>DatetimeIndex.week</code>	The week ordinal of the year
<code>DatetimeIndex.dayofweek</code>	The day of the week with Monday=0, Sunday=6
<code>DatetimeIndex.weekday</code>	The day of the week with Monday=0, Sunday=6
<code>DatetimeIndex.quarter</code>	The quarter of the date
<code>DatetimeIndex.tz</code>	
<code>DatetimeIndex.freq</code>	Return the frequency object if it is set, otherwise None
<code>DatetimeIndex.freqstr</code>	Return the frequency object as a string if it is set, otherwise None
<code>DatetimeIndex.is_month_start</code>	Logical indicating if first day of month (defined by frequency)
<code>DatetimeIndex.is_month_end</code>	Indicator for whether the date is the last day of the month.
<code>DatetimeIndex.is_quarter_start</code>	Indicator for whether the date is the first day of a quarter.
<code>DatetimeIndex.is_quarter_end</code>	Indicator for whether the date is the last day of a quarter.
<code>DatetimeIndex.is_year_start</code>	Indicate whether the date is the first day of a year.
<code>DatetimeIndex.is_year_end</code>	Indicate whether the date is the last day of the year.
<code>DatetimeIndex.is_leap_year</code>	Boolean indicator if the date belongs to a leap year.
<code>DatetimeIndex.inferred_freq</code>	Tries to return a string representing a frequency guess, generated by infer_freq.

### 34.11.2.1 pandas.DatetimeIndex.tz

`DatetimeIndex.tz`

## 34.11.3 Selecting

<code>DatetimeIndex.indexer_at_time(time[, asof])</code>	Returns index locations of index values at particular time of day (e.g.
<code>DatetimeIndex.indexer_between_time(...[, ...])</code>	Return index locations of values between particular times of day (e.g., 9:00-9:30AM).

### 34.11.3.1 pandas.DatetimeIndex.indexer\_at\_time

`DatetimeIndex.indexer_at_time` (*time*, *asof*=False)

Returns index locations of index values at particular time of day (e.g. 9:30AM).

**Parameters** *time* : datetime.time or string

datetime.time or string in appropriate format (“%H:%M”, “%H%M”, “%I:%M%p”, “%I%M%p”, “%H:%M:%S”, “%H%M%S”, “%I:%M:%S%p”, “%I%M%S%p”).

**Returns**

**values\_at\_time** [array of integers]

See also:

*indexer\_between\_time, DataFrame.at\_time*

### 34.11.3.2 pandas.DatetimeIndex.indexer\_between\_time

`DatetimeIndex.indexer_between_time` (*start\_time*, *end\_time*, *include\_start=True*, *include\_end=True*)

Return index locations of values between particular times of day (e.g., 9:00-9:30AM).

**Parameters** *start\_time, end\_time* : `datetime.time`, `str`

`datetime.time` or string in appropriate format (“%H:%M”, “%H%M”, “%I:%M%p”, “%I%M%p”, “%H:%M:%S”, “%H%M%S”, “%I:%M:%S%p”, “%I%M%S%p”).

**include\_start** [boolean, default True]

**include\_end** [boolean, default True]

**Returns**

**values\_between\_time** [array of integers]

See also:

*indexer\_at\_time, DataFrame.between\_time*

### 34.11.4 Time-specific operations

<code>DatetimeIndex.normalize()</code>	Convert times to midnight.
<code>DatetimeIndex.strftime(date_format)</code>	Convert to Index using specified <i>date_format</i> .
<code>DatetimeIndex.snap([freq])</code>	Snap time stamps to nearest occurring frequency
<code>DatetimeIndex.tz_convert(tz)</code>	Convert tz-aware <code>DatetimeIndex</code> from one time zone to another.
<code>DatetimeIndex.tz_localize(tz[, ambiguous, ...])</code>	Localize tz-naive <code>DatetimeIndex</code> to tz-aware <code>Date-timeIndex</code> .
<code>DatetimeIndex.round(freq, *args, **kwargs)</code>	round the data to the specified <i>freq</i> .
<code>DatetimeIndex.floor(freq)</code>	floor the data to the specified <i>freq</i> .
<code>DatetimeIndex.ceil(freq)</code>	ceil the data to the specified <i>freq</i> .
<code>DatetimeIndex.month_name([locale])</code>	Return the month names of the <code>Date-timeIndex</code> with specified locale.
<code>DatetimeIndex.day_name([locale])</code>	Return the day names of the <code>Date-timeIndex</code> with specified locale.

### 34.11.5 Conversion

<code>DatetimeIndex.to_period([freq])</code>	Cast to <code>PeriodIndex</code> at a particular frequency.
<code>DatetimeIndex.to_perioddelta(freq)</code>	Calculate <code>TimedeltaIndex</code> of difference between index values and index converted to <code>periodIndex</code> at specified <i>freq</i> .

Continued on next page

Table 130 – continued from previous page

<code>DatetimeIndex.to_pydatetime()</code>	Return <code>DatetimeIndex</code> as object ndarray of date-time.datetime objects
<code>DatetimeIndex.to_series([keep_tz, index, name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
<code>DatetimeIndex.to_frame([index])</code>	Create a DataFrame with a column containing the Index.

## 34.12 TimedeltaIndex

<code>TimedeltaIndex</code>	Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects
-----------------------------	---

### 34.12.1 pandas.TimedeltaIndex

**class** `pandas.TimedeltaIndex`

Immutable ndarray of timedelta64 data, represented internally as int64, and which can be boxed to timedelta objects

**Parameters** `data` : array-like (1-dimensional), optional

Optional timedelta-like data to construct index with

**unit**: unit of the arg (`D,h,m,s,ms,us,ns`) denote the unit, optional

which is an integer/float number

**freq**: a frequency for the index, optional

**copy** : bool

Make a copy of input ndarray

**start** : starting value, timedelta-like, optional

If data is None, start is used as the start point in generating regular timedelta data.

**periods** : int, optional, > 0

Number of periods to generate, if generating index. Takes precedence over end argument

**end** : end time, timedelta-like, optional

If periods is none, generated index will extend to first conforming time on or just past end argument

**closed** : string or None, default None

Make the interval closed with respect to the given frequency to the 'left', 'right', or both sides (None)

**name** : object

Name to be stored in the index

See also:

**Index** The base pandas Index type

**Timedelta** Represents a duration between two dates or times.

**DatetimeIndex** Index of datetime64 data

**PeriodIndex** Index of Period data

## Notes

To learn more about the frequency strings, please see [this link](#).

## Attributes

<code>days</code>	Number of days for each element.
<code>seconds</code>	Number of seconds ( $\geq 0$ and less than 1 day) for each element.
<code>microseconds</code>	Number of microseconds ( $\geq 0$ and less than 1 second) for each element.
<code>nanoseconds</code>	Number of nanoseconds ( $\geq 0$ and less than 1 microsecond) for each element.
<code>components</code>	Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.
<code>inferred_freq</code>	Tries to return a string representing a frequency guess, generated by <code>infer_freq</code> .

### 34.12.1.1 pandas.TimedeltaIndex.days

`TimedeltaIndex.days`

Number of days for each element.

### 34.12.1.2 pandas.TimedeltaIndex.seconds

`TimedeltaIndex.seconds`

Number of seconds ( $\geq 0$  and less than 1 day) for each element.

### 34.12.1.3 pandas.TimedeltaIndex.microseconds

`TimedeltaIndex.microseconds`

Number of microseconds ( $\geq 0$  and less than 1 second) for each element.

### 34.12.1.4 pandas.TimedeltaIndex.nanoseconds

`TimedeltaIndex.nanoseconds`

Number of nanoseconds ( $\geq 0$  and less than 1 microsecond) for each element.

### 34.12.1.5 pandas.TimedeltaIndex.components

`TimedeltaIndex.components`

Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.

**Returns**

a **DataFrame**

### 34.12.1.6 pandas.TimedeltaIndex.inferred\_freq

`TimedeltaIndex.inferred_freq`

Tries to return a string representing a frequency guess, generated by `infer_freq`. Returns `None` if it can't autodetect the frequency.

#### Methods

<code>to_pytimedelta()</code>	Return <code>TimedeltaIndex</code> as object ndarray of <code>datetime.timedelta</code> objects
<code>to_series([index, name])</code>	Create a <code>Series</code> with both index and values equal to the index keys useful with <code>map</code> for returning an indexer based on an index
<code>round(freq, *args, **kwargs)</code>	round the data to the specified <i>freq</i> .
<code>floor(freq)</code>	floor the data to the specified <i>freq</i> .
<code>ceil(freq)</code>	ceil the data to the specified <i>freq</i> .
<code>to_frame([index])</code>	Create a <code>DataFrame</code> with a column containing the Index.

### 34.12.1.7 pandas.TimedeltaIndex.to\_pytimedelta

`TimedeltaIndex.to_pytimedelta()`

Return `TimedeltaIndex` as object ndarray of `datetime.timedelta` objects

**Returns**

**datetimes** [ndarray]

### 34.12.1.8 pandas.TimedeltaIndex.to\_series

`TimedeltaIndex.to_series(index=None, name=None)`

Create a `Series` with both index and values equal to the index keys useful with `map` for returning an indexer based on an index

**Parameters** `index` : Index, optional

index of resulting `Series`. If `None`, defaults to original index

`name` : string, optional

name of resulting `Series`. If `None`, defaults to name of original index

**Returns**

**Series** [dtype will be based on the type of the Index values.]

### 34.12.1.9 pandas.TimedeltaIndex.round

`TimedeltaIndex.round(freq, *args, **kwargs)`  
round the data to the specified *freq*.

**Parameters** *freq* : str or Offset

The frequency level to round the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** `DatetimeIndex`, `TimedeltaIndex`, or `Series`

Index of the same type for a `DatetimeIndex` or `TimedeltaIndex`, or a `Series` with the same index for a `Series`.

**Raises**

**ValueError** if the ‘freq’ cannot be converted.

#### Examples

##### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.round('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

##### Series

```
>>> pd.Series(rng).dt.round("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

### 34.12.1.10 pandas.TimedeltaIndex.floor

`TimedeltaIndex.floor(freq)`  
floor the data to the specified *freq*.

**Parameters** *freq* : str or Offset

The frequency level to floor the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** `DatetimeIndex`, `TimedeltaIndex`, or `Series`

Index of the same type for a `DatetimeIndex` or `TimedeltaIndex`, or a `Series` with the same index for a `Series`.

**Raises**

**ValueError if the ‘freq’ cannot be converted.**

## Examples

### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.floor('H')
DatetimeIndex(['2018-01-01 11:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

### Series

```
>>> pd.Series(rng).dt.floor("H")
0    2018-01-01 11:00:00
1    2018-01-01 12:00:00
2    2018-01-01 12:00:00
dtype: datetime64[ns]
```

#### 34.12.1.11 pandas.TimedeltaIndex.ceil

`TimedeltaIndex.ceil(freq)`

ceil the data to the specified *freq*.

**Parameters** *freq* : str or Offset

The frequency level to ceil the index to. Must be a fixed frequency like ‘S’ (second) not ‘ME’ (month end). See [frequency aliases](#) for a list of possible *freq* values.

**Returns** `DatetimeIndex`, `TimedeltaIndex`, or `Series`

Index of the same type for a `DatetimeIndex` or `TimedeltaIndex`, or a `Series` with the same index for a `Series`.

**Raises**

**ValueError** if the ‘freq’ cannot be converted.

## Examples

### DatetimeIndex

```
>>> rng = pd.date_range('1/1/2018 11:59:00', periods=3, freq='min')
>>> rng
DatetimeIndex(['2018-01-01 11:59:00', '2018-01-01 12:00:00',
               '2018-01-01 12:01:00'],
              dtype='datetime64[ns]', freq='T')
>>> rng.ceil('H')
DatetimeIndex(['2018-01-01 12:00:00', '2018-01-01 12:00:00',
               '2018-01-01 12:00:00'],
              dtype='datetime64[ns]', freq=None)
```

(continues on next page)

(continued from previous page)

```
'2018-01-01 13:00:00'],
dtype='datetime64[ns]', freq=None)
```

**Series**

```
>>> pd.Series(rng).dt.ceil("H")
0    2018-01-01 12:00:00
1    2018-01-01 12:00:00
2    2018-01-01 13:00:00
dtype: datetime64[ns]
```

**34.12.1.12 pandas.TimedeltaIndex.to\_frame****TimedeltaIndex.to\_frame** (*index=True*)

Create a DataFrame with a column containing the Index.

New in version 0.21.0.

**Parameters** *index* : boolean, default True

Set the index of the returned DataFrame as the original Index.

**Returns** **DataFrame**

DataFrame containing the original Index data.

**See also:*****Index.to\_series*** Convert an Index to a Series.***Series.to\_frame*** Convert Series to DataFrame.**Examples**

```
>>> idx = pd.Index(['Ant', 'Bear', 'Cow'], name='animal')
>>> idx.to_frame()
      animal
animal
Ant      Ant
Bear    Bear
Cow     Cow
```

By default, the original Index is reused. To enforce a new Index:

```
>>> idx.to_frame(index=False)
      animal
0      Ant
1     Bear
2      Cow
```

**34.12.2 Components**



<code>TimedeltaIndex.days</code>	Number of days for each element.
<code>TimedeltaIndex.seconds</code>	Number of seconds ( $\geq 0$ and less than 1 day) for each element.
<code>TimedeltaIndex.microseconds</code>	Number of microseconds ( $\geq 0$ and less than 1 second) for each element.
<code>TimedeltaIndex.nanoseconds</code>	Number of nanoseconds ( $\geq 0$ and less than 1 microsecond) for each element.
<code>TimedeltaIndex.components</code>	Return a dataframe of the components (days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds) of the Timedeltas.
<code>TimedeltaIndex.inferred_freq</code>	Tries to return a string representing a frequency guess, generated by <code>infer_freq</code> .

### 34.12.3 Conversion

<code>TimedeltaIndex.to_pytimedelta()</code>	Return <code>TimedeltaIndex</code> as object ndarray of date-time.timedelta objects
<code>TimedeltaIndex.to_series([index, name])</code>	Create a Series with both index and values equal to the index keys useful with map for returning an indexer based on an index
<code>TimedeltaIndex.round(freq, *args, **kwargs)</code>	round the data to the specified <i>freq</i> .
<code>TimedeltaIndex.floor(freq)</code>	floor the data to the specified <i>freq</i> .
<code>TimedeltaIndex.ceil(freq)</code>	ceil the data to the specified <i>freq</i> .
<code>TimedeltaIndex.to_frame([index])</code>	Create a DataFrame with a column containing the Index.

## 34.13 PeriodIndex

<code>PeriodIndex</code>	Immutable ndarray holding ordinal values indicating regular periods in time such as particular years, quarters, months, etc.
--------------------------	--

### 34.13.1 pandas.PeriodIndex

**class** pandas.**PeriodIndex**

Immutable ndarray holding ordinal values indicating regular periods in time such as particular years, quarters, months, etc.

Index keys are boxed to Period objects which carries the metadata (eg, frequency information).

**Parameters** **data** : array-like (1-dimensional), optional

Optional period-like data to construct index with

**copy** : bool

Make a copy of input ndarray

**freq** : string or period object, optional

One of pandas period strings or corresponding objects

**start** : starting value, period-like, optional

If data is None, used as the start point in generating regular period data.

**periods** : int, optional, > 0

Number of periods to generate, if generating index. Takes precedence over end argument

**end** : end value, period-like, optional

If periods is none, generated index will extend to first conforming period on or just past end argument

**year** [int, array, or Series, default None]

**month** [int, array, or Series, default None]

**quarter** [int, array, or Series, default None]

**day** [int, array, or Series, default None]

**hour** [int, array, or Series, default None]

**minute** [int, array, or Series, default None]

**second** [int, array, or Series, default None]

**tz** : object, default None

Timezone for converting datetime64 data to Periods

**dtype** [str or PeriodDtype, default None]

**See also:**

***Index*** The base pandas Index type

***Period*** Represents a period of time

***DatetimeIndex*** Index with datetime64 data

***TimedeltaIndex*** Index of timedelta64 data

## Examples

```
>>> idx = PeriodIndex(year=year_arr, quarter=q_arr)
```

```
>>> idx2 = PeriodIndex(start='2000', end='2010', freq='A')
```

## Attributes

<i>day</i>	The days of the period
<i>dayofweek</i>	The day of the week with Monday=0, Sunday=6
<i>dayofyear</i>	The ordinal day of the year
<i>days_in_month</i>	The number of days in the month
<i>daysinmonth</i>	The number of days in the month
<i>freq</i>	Return the frequency object if it is set, otherwise None

Continued on next page

Table 137 – continued from previous page

<i>freqstr</i>	Return the frequency object as a string if it is set, otherwise None
<i>hour</i>	The hour of the period
<i>is_leap_year</i>	Logical indicating if the date belongs to a leap year
<i>minute</i>	The minute of the period
<i>month</i>	The month as January=1, December=12
<i>quarter</i>	The quarter of the date
<i>second</i>	The second of the period
<i>week</i>	The week ordinal of the year
<i>weekday</i>	The day of the week with Monday=0, Sunday=6
<i>weekofyear</i>	The week ordinal of the year
<i>year</i>	The year of the period

**34.13.1.1 pandas.PeriodIndex.day**

`PeriodIndex.day`  
The days of the period

**34.13.1.2 pandas.PeriodIndex.dayofweek**

`PeriodIndex.dayofweek`  
The day of the week with Monday=0, Sunday=6

**34.13.1.3 pandas.PeriodIndex.dayofyear**

`PeriodIndex.dayofyear`  
The ordinal day of the year

**34.13.1.4 pandas.PeriodIndex.days\_in\_month**

`PeriodIndex.days_in_month`  
The number of days in the month

**34.13.1.5 pandas.PeriodIndex.daysinmonth**

`PeriodIndex.daysinmonth`  
The number of days in the month

**34.13.1.6 pandas.PeriodIndex.freq**

`PeriodIndex.freq`  
Return the frequency object if it is set, otherwise None

**34.13.1.7 pandas.PeriodIndex.freqstr**

`PeriodIndex.freqstr`  
Return the frequency object as a string if it is set, otherwise None

#### 34.13.1.8 pandas.PeriodIndex.hour

`PeriodIndex.hour`  
The hour of the period

#### 34.13.1.9 pandas.PeriodIndex.is\_leap\_year

`PeriodIndex.is_leap_year`  
Logical indicating if the date belongs to a leap year

#### 34.13.1.10 pandas.PeriodIndex.minute

`PeriodIndex.minute`  
The minute of the period

#### 34.13.1.11 pandas.PeriodIndex.month

`PeriodIndex.month`  
The month as January=1, December=12

#### 34.13.1.12 pandas.PeriodIndex.quarter

`PeriodIndex.quarter`  
The quarter of the date

#### 34.13.1.13 pandas.PeriodIndex.second

`PeriodIndex.second`  
The second of the period

#### 34.13.1.14 pandas.PeriodIndex.week

`PeriodIndex.week`  
The week ordinal of the year

#### 34.13.1.15 pandas.PeriodIndex.weekday

`PeriodIndex.weekday`  
The day of the week with Monday=0, Sunday=6

#### 34.13.1.16 pandas.PeriodIndex.weekofyear

`PeriodIndex.weekofyear`  
The week ordinal of the year

### 34.13.1.17 pandas.PeriodIndex.year

`PeriodIndex.year`

The year of the period

<b>end_time</b>	
<b>qyear</b>	
<b>start_time</b>	

#### Methods

<code>asfreq([freq, how])</code>	Convert the PeriodIndex to the specified frequency <i>freq</i> .
<code>strftime(date_format)</code>	Convert to Index using specified <i>date_format</i> .
<code>to_timestamp([freq, how])</code>	Cast to DatetimeIndex
<code>tz_convert(tz)</code>	Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)
<code>tz_localize(tz[, ambiguous])</code>	Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil), or remove timezone from tz-aware DatetimeIndex

### 34.13.1.18 pandas.PeriodIndex.asfreq

`PeriodIndex.asfreq(freq=None, how='E')`

Convert the PeriodIndex to the specified frequency *freq*.

**Parameters** *freq* : str

a frequency

**how** : str {'E', 'S'}

'E', 'END', or 'FINISH' for end, 'S', 'START', or 'BEGIN' for start. Whether the elements should be aligned to the end or start within a period. January 31st ('END') vs. January 1st ('START') for example.

**Returns**

**new** [PeriodIndex with the new frequency]

#### Examples

```
>>> pidx = pd.period_range('2010-01-01', '2015-01-01', freq='A')
>>> pidx
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010, ..., 2015]
Length: 6, Freq: A-DEC
```

```
>>> pidx.asfreq('M')
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010-12, ..., 2015-12]
Length: 6, Freq: M
```

```
>>> pidx.asfreq('M', how='S')
<class 'pandas.core.indexes.period.PeriodIndex'>
[2010-01, ..., 2015-01]
Length: 6, Freq: M
```

### 34.13.1.19 pandas.PeriodIndex.strftime

`PeriodIndex.strftime(date_format)`

Convert to Index using specified date\_format.

Return an Index of formatted strings specified by date\_format, which supports the same string format as the python standard library. Details of the string format can be found in [python string format doc](#)

**Parameters** `date_format` : str

Date format string (e.g. “%Y-%m-%d”).

**Returns** Index

Index of formatted strings

**See also:**

[`pandas.to\_datetime`](#) Convert the given argument to datetime

[`DatetimeIndex.normalize`](#) Return DatetimeIndex with times to midnight.

[`DatetimeIndex.round`](#) Round the DatetimeIndex to the specified freq.

[`DatetimeIndex.floor`](#) Floor the DatetimeIndex to the specified freq.

### Examples

```
>>> rng = pd.date_range(pd.Timestamp("2018-03-10 09:00"),
...                     periods=3, freq='s')
>>> rng.strftime('%B %d, %Y, %r')
Index(['March 10, 2018, 09:00:00 AM', 'March 10, 2018, 09:00:01 AM',
       'March 10, 2018, 09:00:02 AM'],
      dtype='object')
```

### 34.13.1.20 pandas.PeriodIndex.to\_timestamp

`PeriodIndex.to_timestamp(freq=None, how='start')`

Cast to DatetimeIndex

**Parameters** `freq` : string or DateOffset, optional

Target frequency. The default is ‘D’ for week or longer, ‘S’ otherwise

**how** [{‘s’, ‘e’, ‘start’, ‘end’}]

**Returns**

DatetimeIndex

### 34.13.1.21 pandas.PeriodIndex.tz\_convert

`PeriodIndex.tz_convert(tz)`

Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)

**Parameters** `tz` : string, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding UTC time.

**Returns**

**normalized** [DatetimeIndex]

#### Notes

Not currently implemented for PeriodIndex

### 34.13.1.22 pandas.PeriodIndex.tz\_localize

`PeriodIndex.tz_localize(tz, ambiguous='raise')`

Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil), or remove timezone from tz-aware DatetimeIndex

**Parameters** `tz` : string, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time. Corresponding timestamps would be converted to time zone of the TimeSeries. None will remove timezone holding local time.

**Returns**

**localized** [DatetimeIndex]

#### Notes

Not currently implemented for PeriodIndex

## 34.13.2 Attributes

<code>PeriodIndex.day</code>	The days of the period
<code>PeriodIndex.dayofweek</code>	The day of the week with Monday=0, Sunday=6
<code>PeriodIndex.dayofyear</code>	The ordinal day of the year
<code>PeriodIndex.days_in_month</code>	The number of days in the month
<code>PeriodIndex.daysinmonth</code>	The number of days in the month
<code>PeriodIndex.end_time</code>	
<code>PeriodIndex.freq</code>	Return the frequency object if it is set, otherwise None
<code>PeriodIndex.freqstr</code>	Return the frequency object as a string if it is set, otherwise None
<code>PeriodIndex.hour</code>	The hour of the period
<code>PeriodIndex.is_leap_year</code>	Logical indicating if the date belongs to a leap year
<code>PeriodIndex.minute</code>	The minute of the period
<code>PeriodIndex.month</code>	The month as January=1, December=12

Continued on next page

Table 139 – continued from previous page

<i>PeriodIndex.quarter</i>	The quarter of the date
<i>PeriodIndex.qyear</i>	
<i>PeriodIndex.second</i>	The second of the period
<i>PeriodIndex.start_time</i>	
<i>PeriodIndex.week</i>	The week ordinal of the year
<i>PeriodIndex.weekday</i>	The day of the week with Monday=0, Sunday=6
<i>PeriodIndex.weekofyear</i>	The week ordinal of the year
<i>PeriodIndex.year</i>	The year of the period

### 34.13.2.1 pandas.PeriodIndex.end\_time

`PeriodIndex.end_time`

### 34.13.2.2 pandas.PeriodIndex.qyear

`PeriodIndex.qyear`

### 34.13.2.3 pandas.PeriodIndex.start\_time

`PeriodIndex.start_time`

## 34.13.3 Methods

<i>PeriodIndex.asfreq</i> ([freq, how])	Convert the PeriodIndex to the specified frequency <i>freq</i> .
<i>PeriodIndex.strftime</i> (date_format)	Convert to Index using specified date_format.
<i>PeriodIndex.to_timestamp</i> ([freq, how])	Cast to DatetimeIndex
<i>PeriodIndex.tz_convert</i> (tz)	Convert tz-aware DatetimeIndex from one time zone to another (using pytz/dateutil)
<i>PeriodIndex.tz_localize</i> (tz[, ambiguous])	Localize tz-naive DatetimeIndex to given time zone (using pytz/dateutil), or remove timezone from tz-aware DatetimeIndex

## 34.14 Scalars

### 34.14.1 Period

<i>Period</i>	Represents a period of time
---------------	-----------------------------

#### 34.14.1.1 pandas.Period

**class** pandas.Period

Represents a period of time

**Parameters** **value** : Period or compat.string\_types, default None

The time period represented (e.g., '4Q2005')

**freq** : str, default None



One of pandas period strings or corresponding objects

**year** [int, default None]  
**month** [int, default 1]  
**quarter** [int, default None]  
**day** [int, default 1]  
**hour** [int, default 0]  
**minute** [int, default 0]  
**second** [int, default 0]

### Attributes

<i>day</i>	Get day of the month that a Period falls on.
<i>dayofweek</i>	Return the day of the week.
<i>dayofyear</i>	Return the day of the year.
<i>days_in_month</i>	Get the total number of days in the month that this period falls on.
<i>daysinmonth</i>	Get the total number of days of the month that the Period falls in.
<i>hour</i>	Get the hour of the day component of the Period.
<i>minute</i>	Get minute of the hour component of the Period.
<i>second</i>	Get the second component of the Period.
<i>start_time</i>	Get the Timestamp for the start of the period.
<i>week</i>	Get the week of the year on the given Period.

### pandas.Period.day

`Period.day`

Get day of the month that a Period falls on.

#### Returns

**int**

**See also:**

*Period.dayofweek* Get the day of the week

*Period.dayofyear* Get the day of the year

### Examples

```
>>> p = pd.Period("2018-03-11", freq='H')
>>> p.day
11
```

## pandas.Period.dayofweek

### Period.dayofweek

Return the day of the week.

This attribute returns the day of the week on which the particular date for the given period occurs depending on the frequency with Monday=0, Sunday=6.

#### Returns Int

Range from 0 to 6 (included).

#### See also:

*Period.dayofyear* Return the day of the year.

*Period.daysinmonth* Return the number of days in that month.

## Examples

```
>>> period1 = pd.Period('2012-1-1 19:00', freq='H')
>>> period1
Period('2012-01-01 19:00', 'H')
>>> period1.dayofweek
6
```

```
>>> period2 = pd.Period('2013-1-9 11:00', freq='H')
>>> period2
Period('2013-01-09 11:00', 'H')
>>> period2.dayofweek
2
```

## pandas.Period.dayofyear

### Period.dayofyear

Return the day of the year.

This attribute returns the day of the year on which the particular date occurs. The return value ranges between 1 to 365 for regular years and 1 to 366 for leap years.

#### Returns int

The day of year.

#### See also:

*Period.day* Return the day of the month.

*Period.dayofweek* Return the day of week.

*PeriodIndex.dayofyear* Return the day of year of all indexes.

## Examples

```

>>> period = pd.Period("2015-10-23", freq='H')
>>> period.dayofyear
296
>>> period = pd.Period("2012-12-31", freq='D')
>>> period.dayofyear
366
>>> period = pd.Period("2013-01-01", freq='D')
>>> period.dayofyear
1

```

## pandas.Period.days\_in\_month

### Period.days\_in\_month

Get the total number of days in the month that this period falls on.

#### Returns

int

#### See also:

**Period.daysinmonth** Gets the number of days in the month.

**DatetimeIndex.daysinmonth** Gets the number of days in the month.

**calendar.monthrange** Returns a tuple containing weekday (0-6 ~ Mon-Sun) and number of days (28-31).

## Examples

```

>>> p = pd.Period('2018-2-17')
>>> p.days_in_month
28

```

```

>>> pd.Period('2018-03-01').days_in_month
31

```

Handles the leap year case as well:

```

>>> p = pd.Period('2016-2-17')
>>> p.days_in_month
29

```

## pandas.Period.daysinmonth

### Period.daysinmonth

Get the total number of days of the month that the Period falls in.

#### Returns

int

#### See also:

*Period.days\_in\_month* Return the days of the month

*Period.dayofyear* Return the day of the year

### Examples

```
>>> p = pd.Period("2018-03-11", freq='H')
>>> p.daysinmonth
31
```

## pandas.Period.hour

**Period.hour**

Get the hour of the day component of the Period.

**Returns int**

The hour as an integer, between 0 and 23.

**See also:**

*Period.second* Get the second component of the Period.

*Period.minute* Get the minute component of the Period.

### Examples

```
>>> p = pd.Period("2018-03-11 13:03:12.050000")
>>> p.hour
13
```

Period longer than a day

```
>>> p = pd.Period("2018-03-11", freq="M")
>>> p.hour
0
```

## pandas.Period.minute

**Period.minute**

Get minute of the hour component of the Period.

**Returns int**

The minute as an integer, between 0 and 59.

**See also:**

*Period.hour* Get the hour component of the Period.

*Period.second* Get the second component of the Period.

## Examples

```
>>> p = pd.Period("2018-03-11 13:03:12.050000")
>>> p.minute
3
```

## pandas.Period.second

Period.**second**

Get the second component of the Period.

**Returns int**

The second of the Period (ranges from 0 to 59).

**See also:**

*Period.hour* Get the hour component of the Period.

*Period.minute* Get the minute component of the Period.

## Examples

```
>>> p = pd.Period("2018-03-11 13:03:12.050000")
>>> p.second
12
```

## pandas.Period.start\_time

Period.**start\_time**

Get the Timestamp for the start of the period.

**Returns**

**Timestamp**

**See also:**

*Period.end\_time* Return the end Timestamp.

*Period.dayofyear* Return the day of year.

*Period.daysinmonth* Return the days in that month.

*Period.dayofweek* Return the day of the week.

## Examples

```
>>> period = pd.Period('2012-1-1', freq='D')
>>> period
Period('2012-01-01', 'D')
```

```
>>> period.start_time
Timestamp('2012-01-01 00:00:00')
```

```
>>> period.end_time
Timestamp('2012-01-01 23:59:59.999999999')
```

## pandas.Period.week

### Period.week

Get the week of the year on the given Period.

#### Returns

**int**

See also:

*Period.dayofweek* Get the day component of the Period.

*Period.weekday* Get the day component of the Period.

### Examples

```
>>> p = pd.Period("2018-03-11", "H")
>>> p.week
10
```

```
>>> p = pd.Period("2018-02-01", "D")
>>> p.week
5
```

```
>>> p = pd.Period("2018-01-06", "D")
>>> p.week
1
```

<b>end_time</b>	
<b>freq</b>	
<b>freqstr</b>	
<b>is_leap_year</b>	
<b>month</b>	
<b>ordinal</b>	
<b>quarter</b>	
<b>qyear</b>	
<b>weekday</b>	
<b>weekofyear</b>	
<b>year</b>	

### Methods

<code>asfreq</code>	Convert Period to desired frequency, either at the start or end of the interval
<code>strftime</code>	Returns the string representation of the <i>Period</i> , depending on the selected <code>fmt</code> .
<code>to_timestamp</code>	Return the Timestamp representation of the Period at the target frequency at the specified end (how) of the Period

**pandas.Period.asfreq**`Period.asfreq()`

Convert Period to desired frequency, either at the start or end of the interval

**Parameters****freq** [string]**how** : { 'E', 'S', 'end', 'start' }, default 'end'

Start or end of the timespan

**Returns****resampled** [Period]**pandas.Period.strftime**`Period.strftime()`

Returns the string representation of the *Period*, depending on the selected `fmt`. `fmt` must be a string containing one or several directives. The method recognizes the same directives as the `time.strftime()` function of the standard Python distribution, as well as the specific additional directives `%f`, `%F`, `%q`. (formatting & docs originally from `scikits.timeries`)

Di- rec- tive	Meaning	Notes
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%f	'Fiscal' year without a century as a decimal number [00,99]	(1)
%F	'Fiscal' year with a century as a decimal number	(2)
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	(3)
%q	Quarter as a decimal number [01,04]	
%S	Second as a decimal number [00,61].	(4)
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	(5)
%w	Weekday as a decimal number [0(Sunday),6].	
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	(5)
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%Z	Time zone name (no characters if no time zone exists).	
%%	A literal '%' character.	

## Notes

1. The %f directive is the same as %y if the frequency is not quarterly. Otherwise, it corresponds to the 'fiscal' year, as defined by the *qyear* attribute.
2. The %F directive is the same as %Y if the frequency is not quarterly. Otherwise, it corresponds to the 'fiscal' year, as defined by the *qyear* attribute.
3. The %p directive only affects the output hour field if the %I directive is used to parse the hour.
4. The range really is 0 to 61; this accounts for leap seconds and the (very rare) double leap seconds.
5. The %U and %W directives are only used in calculations when the day of the week and the year are specified.

## Examples



```
>>> a = Period(freq='Q-JUL', year=2006, quarter=1)
>>> a.strftime('%F-Q%q')
'2006-Q1'
>>> # Output the last month in the quarter of this date
>>> a.strftime('%b-%Y')
'Oct-2005'
>>>
>>> a = Period(freq='D', year=2001, month=1, day=1)
>>> a.strftime('%d-%b-%Y')
'01-Jan-2006'
>>> a.strftime('%b. %d, %Y was a %A')
'Jan. 01, 2001 was a Monday'
```

pandas.Period.to\_timestamp

`Period.to_timestamp()`  
Return the Timestamp representation of the Period at the target frequency at the specified end (how) of the Period

- Parameters** **freq** : string or DateOffset  
Target frequency. Default is 'D' if self.freq is week or longer and 'S' otherwise
- how: str, default 'S' (start)**  
'S', 'E'. Can be aliased as case insensitive 'Start', 'Finish', 'Begin', 'End'

**Returns**  
**Timestamp**

now	
-----	--

34.14.2 Attributes

<code>Period.day</code>	Get day of the month that a Period falls on.
<code>Period.dayofweek</code>	Return the day of the week.
<code>Period.dayofyear</code>	Return the day of the year.
<code>Period.days_in_month</code>	Get the total number of days in the month that this period falls on.
<code>Period.daysinmonth</code>	Get the total number of days of the month that the Period falls in.
<code>Period.end_time</code>	
<code>Period.freq</code>	
<code>Period.freqstr</code>	
<code>Period.hour</code>	Get the hour of the day component of the Period.
<code>Period.is_leap_year</code>	
<code>Period.minute</code>	Get minute of the hour component of the Period.
<code>Period.month</code>	
<code>Period.ordinal</code>	
<code>Period.quarter</code>	
<code>Period.qyear</code>	

Continued on next page

Table 144 – continued from previous page

<i>Period.second</i>	Get the second component of the Period.
<i>Period.start_time</i>	Get the Timestamp for the start of the period.
<i>Period.week</i>	Get the week of the year on the given Period.
<i>Period.weekday</i>	
<i>Period.weekofyear</i>	
<i>Period.year</i>	

#### 34.14.2.1 pandas.Period.end\_time

Period.**end\_time**

#### 34.14.2.2 pandas.Period.freq

Period.**freq**

#### 34.14.2.3 pandas.Period.freqstr

Period.**freqstr**

#### 34.14.2.4 pandas.Period.is\_leap\_year

Period.**is\_leap\_year**

#### 34.14.2.5 pandas.Period.month

Period.**month**

#### 34.14.2.6 pandas.Period.ordinal

Period.**ordinal**

#### 34.14.2.7 pandas.Period.quarter

Period.**quarter**

#### 34.14.2.8 pandas.Period.qyear

Period.**qyear**

#### 34.14.2.9 pandas.Period.weekday

Period.**weekday**

#### 34.14.2.10 pandas.Period.weekofyear

Period.**weekofyear**

### 34.14.2.11 pandas.Period.year

Period.**year**

## 34.14.3 Methods

<i>Period.asfreq</i>	Convert Period to desired frequency, either at the start or end of the interval
<i>Period.now</i>	
<i>Period.strftime</i>	Returns the string representation of the <i>Period</i> , depending on the selected <i>fmt</i> .
<i>Period.to_timestamp</i>	Return the Timestamp representation of the Period at the target frequency at the specified end (how) of the Period

### 34.14.3.1 pandas.Period.now

Period.**now**()

## 34.14.4 Timestamp

<i>Timestamp</i>	Pandas replacement for datetime.datetime
------------------	--

### 34.14.4.1 pandas.Timestamp

**class** pandas.**Timestamp**

Pandas replacement for datetime.datetime

Timestamp is the pandas equivalent of python's Datetime and is interchangeable with it in most cases. It's the type used for the entries that make up a DatetimeIndex, and other timeseries oriented data structures in pandas.

**Parameters** **ts\_input** : datetime-like, str, int, float

Value to be converted to Timestamp

**freq** : str, DateOffset

Offset which Timestamp will have

**tz** : str, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time which Timestamp will have.

**unit** : str

Unit used for conversion if ts\_input is of type int or float. The valid values are 'D', 'h', 'm', 's', 'ms', 'us', and 'ns'. For example, 's' means seconds and 'ms' means milliseconds.

**year, month, day** : int

New in version 0.19.0.

**hour, minute, second, microsecond** : int, optional, default 0

New in version 0.19.0.

**nanosecond** : int, optional, default 0

New in version 0.23.0.

**tzinfo** : datetime.tzinfo, optional, default None

New in version 0.19.0.

## Notes

There are essentially three calling conventions for the constructor. The primary form accepts four parameters. They can be passed by position or keyword.

The other two forms mimic the parameters from `datetime.datetime`. They can be passed by either position or keyword, but not both mixed together.

## Examples

Using the primary calling convention:

This converts a datetime-like string >>> `pd.Timestamp('2017-01-01T12')` `Timestamp('2017-01-01 12:00:00')`

This converts a float representing a Unix epoch in units of seconds >>> `pd.Timestamp(1513393355.5, unit='s')` `Timestamp('2017-12-16 03:02:35.500000')`

This converts an int representing a Unix-epoch in units of seconds and for a particular time-zone >>> `pd.Timestamp(1513393355, unit='s', tz='US/Pacific')` `Timestamp('2017-12-15 19:02:35-0800', tz='US/Pacific')`

Using the other two forms that mimic the API for `datetime.datetime`:

```
>>> pd.Timestamp(2017, 1, 1, 12)
Timestamp('2017-01-01 12:00:00')
```

```
>>> pd.Timestamp(year=2017, month=1, day=1, hour=12)
Timestamp('2017-01-01 12:00:00')
```

## Attributes

<i>tz</i>	Alias for <code>tzinfo</code>
<i>weekday_name</i>	(DEPRECATED) ..

### **pandas.Timestamp.tz**

`Timestamp.tz`  
Alias for `tzinfo`

### **pandas.Timestamp.weekday\_name**

`Timestamp.weekday_name`  
Deprecated since version 0.23.0: Use `Timestamp.day_name()` instead

asm8	
day	
dayofweek	
dayofyear	
days_in_month	
daysinmonth	
fold	
freq	
freqstr	
hour	
is_leap_year	
is_month_end	
is_month_start	
is_quarter_end	
is_quarter_start	
is_year_end	
is_year_start	
microsecond	
minute	
month	
nanosecond	
quarter	
second	
tzinfo	
value	
week	
weekofyear	
year	

## Methods

<i>astimezone</i>	Convert tz-aware Timestamp to another time zone.
<i>ceil</i>	return a new Timestamp ceiled to this resolution
<i>combine</i> (date, time)	date, time -> datetime with same date and time fields
<i>ctime</i>	Return ctime() style string.
<i>date</i>	Return date object with same year, month and day.
<i>day_name</i>	Return the day name of the Timestamp with specified locale.
<i>dst</i>	Return self.tzinfo.dst(self).
<i>floor</i>	return a new Timestamp floored to this resolution
<i>fromordinal</i> (ordinal[, freq, tz])	passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself
<i>fromtimestamp</i> (ts)	timestamp[, tz] -> tz's local time from POSIX timestamp.
<i>isocalendar</i>	Return a 3-tuple containing ISO year, week number, and weekday.
<i>isoweekday</i>	Return the day of the week represented by the date.

Continued on next page

Table 148 – continued from previous page

<i>month_name</i>	Return the month name of the Timestamp with specified locale.
<i>normalize</i>	Normalize Timestamp to midnight, preserving tz information.
<i>now([tz])</i>	Returns new Timestamp object representing current time local to tz.
<i>replace</i>	implements datetime.replace, handles nanoseconds
<i>round</i>	Round the Timestamp to the specified resolution
<i>strftime</i>	format -> strftime() style string.
<i>strptime</i>	string, format -> new datetime parsed from a string (like time.strptime()).
<i>time</i>	Return time object with same time but with tzinfo=None.
<i>timestamp</i>	Return POSIX timestamp as float.
<i>timetuple</i>	Return time tuple, compatible with time.localtime().
<i>timetz</i>	Return time object with same time and tzinfo.
<i>to_datetime64</i>	Returns a numpy.datetime64 object with 'ns' precision
<i>to_julian_date</i>	Convert TimeStamp to a Julian Date.
<i>to_period</i>	Return an period of which this timestamp is an observation.
<i>to_pydatetime</i>	Convert a Timestamp object to a native Python datetime object.
<i>today(cls[, tz])</i>	Return the current time in the local timezone.
<i>toordinal</i>	Return proleptic Gregorian ordinal.
<i>tz_convert</i>	Convert tz-aware Timestamp to another time zone.
<i>tz_localize</i>	Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.
<i>tzname</i>	Return self.tzinfo.tzname(self).
<i>utcfromtimestamp(ts)</i>	Construct a naive UTC datetime from a POSIX timestamp.
<i>utcnow()</i>	Return a new Timestamp representing UTC day and time.
<i>utcoffset</i>	Return self.tzinfo.utcoffset(self).
<i>utctimetuple</i>	Return UTC time tuple, compatible with time.localtime().
<i>weekday</i>	Return the day of the week represented by the date.

**pandas.Timestamp.astimezone****Timestamp.astimezone**

Convert tz-aware Timestamp to another time zone.

**Parameters** *tz* : str, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

**Returns****converted** [Timestamp]**Raises** **TypeError**

If Timestamp is tz-naive.

**pandas.Timestamp.ceil**

`Timestamp.ceil`  
return a new Timestamp ceiled to this resolution

**Parameters**

**freq** [a freq string indicating the ceiling resolution]

**pandas.Timestamp.combine**

**classmethod** `Timestamp.combine(date, time)`  
date, time -> datetime with same date and time fields

**pandas.Timestamp.ctime**

`Timestamp.ctime()`  
Return ctime() style string.

**pandas.Timestamp.date**

`Timestamp.date()`  
Return date object with same year, month and day.

**pandas.Timestamp.day\_name**

`Timestamp.day_name`  
Return the day name of the Timestamp with specified locale.

**Parameters** **locale** : string, default None (English locale)

locale determining the language in which to return the day name

**Returns**

**day\_name** [string]

.. versionadded:: 0.23.0

**pandas.Timestamp.dst**

`Timestamp.dst()`  
Return self.tzinfo.dst(self).

**pandas.Timestamp.floor**

`Timestamp.floor`  
return a new Timestamp floored to this resolution

**Parameters**

**freq** [a freq string indicating the flooring resolution]

### **pandas.Timestamp.fromordinal**

**classmethod** `Timestamp.fromordinal` (*ordinal*, *freq=None*, *tz=None*)

passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself

**Parameters** **ordinal** : int

date corresponding to a proleptic Gregorian ordinal

**freq** : str, DateOffset

Offset which Timestamp will have

**tz** : str, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time which Timestamp will have.

### **pandas.Timestamp.fromtimestamp**

**classmethod** `Timestamp.fromtimestamp` (*ts*)

timestamp[, tz] -> tz's local time from POSIX timestamp.

### **pandas.Timestamp.isocalendar**

`Timestamp.isocalendar()`

Return a 3-tuple containing ISO year, week number, and weekday.

### **pandas.Timestamp.isoweekday**

`Timestamp.isoweekday()`

Return the day of the week represented by the date. Monday == 1 ... Sunday == 7

### **pandas.Timestamp.month\_name**

`Timestamp.month_name`

Return the month name of the Timestamp with specified locale.

**Parameters** **locale** : string, default None (English locale)

locale determining the language in which to return the month name

**Returns**

**month\_name** [string]

.. versionadded:: 0.23.0

### **pandas.Timestamp.normalize**

`Timestamp.normalize`

Normalize Timestamp to midnight, preserving tz information.



### **pandas.Timestamp.now**

**classmethod** `Timestamp.now` (*tz=None*)

Returns new Timestamp object representing current time local to tz.

**Parameters** *tz* : str or timezone object, default None

Timezone to localize to

### **pandas.Timestamp.replace**

`Timestamp.replace`

implements datetime.replace, handles nanoseconds

**Parameters**

**year** [int, optional]

**month** [int, optional]

**day** [int, optional]

**hour** [int, optional]

**minute** [int, optional]

**second** [int, optional]

**microsecond** [int, optional]

**nanosecond**: int, optional

**tzinfo** [tz-convertible, optional]

**fold** : int, optional, default is 0

added in 3.6, NotImplemented

**Returns**

**Timestamp with fields replaced**

### **pandas.Timestamp.round**

`Timestamp.round`

Round the Timestamp to the specified resolution

**Parameters**

**freq** [a freq string indicating the rounding resolution]

**Returns**

**a new Timestamp rounded to the given resolution of ‘freq’**

**Raises**

**ValueError** if the freq cannot be converted

### **pandas.Timestamp.strftime**

`Timestamp.strftime()`  
format -> strftime() style string.

### **pandas.Timestamp.strptime**

`Timestamp.strptime()`  
string, format -> new datetime parsed from a string (like `time.strptime()`).

### **pandas.Timestamp.time**

`Timestamp.time()`  
Return time object with same time but with `tzinfo=None`.

### **pandas.Timestamp.timestamp**

`Timestamp.timestamp()`  
Return POSIX timestamp as float.

### **pandas.Timestamp.timetuple**

`Timestamp.timetuple()`  
Return time tuple, compatible with `time.localtime()`.

### **pandas.Timestamp.timetz**

`Timestamp.timetz()`  
Return time object with same time and `tzinfo`.

### **pandas.Timestamp.to\_datetime64**

`Timestamp.to_datetime64()`  
Returns a `numpy.datetime64` object with 'ns' precision

### **pandas.Timestamp.to\_julian\_date**

`Timestamp.to_julian_date`  
Convert `TimeStamp` to a Julian Date. 0 Julian date is noon January 1, 4713 BC.

### **pandas.Timestamp.to\_period**

`Timestamp.to_period`  
Return an period of which this timestamp is an observation.

**pandas.Timestamp.to\_pydatetime**`Timestamp.to_pydatetime()`

Convert a Timestamp object to a native Python datetime object.

If warn=True, issue a warning if nanoseconds is nonzero.

**pandas.Timestamp.today****classmethod** `Timestamp.today(cls, tz=None)`

Return the current time in the local timezone. This differs from `datetime.today()` in that it can be localized to a passed timezone.

**Parameters** `tz` : str or timezone object, default None

Timezone to localize to

**pandas.Timestamp.toordinal**`Timestamp.toordinal()`

Return proleptic Gregorian ordinal. January 1 of year 1 is day 1.

**pandas.Timestamp.tz\_convert**`Timestamp.tz_convert`

Convert tz-aware Timestamp to another time zone.

**Parameters** `tz` : str, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time which Timestamp will be converted to. None will remove timezone holding UTC time.

**Returns**

**converted** [Timestamp]

**Raises** `TypeError`

If Timestamp is tz-naive.

**pandas.Timestamp.tz\_localize**`Timestamp.tz_localize`

Convert naive Timestamp to local time zone, or remove timezone from tz-aware Timestamp.

**Parameters** `tz` : str, pytz.timezone, dateutil.tz.tzfile or None

Time zone for time which Timestamp will be converted to. None will remove timezone holding local time.

**ambiguous** : bool, 'NaT', default 'raise'

- bool contains flags to determine if time is dst or not (note that this flag is only applicable for ambiguous fall dst dates)
- 'NaT' will return NaT for an ambiguous time

- ‘raise’ will raise an `AmbiguousTimeError` for an ambiguous time

**errors** : ‘raise’, ‘coerce’, default ‘raise’

- ‘raise’ will raise a `NonExistentTimeError` if a timestamp is not valid in the specified timezone (e.g. due to a transition from or to DST time)
- ‘coerce’ will return `NaT` if the timestamp can not be converted into the specified timezone

New in version 0.19.0.

### Returns

**localized** [Timestamp]

### Raises `TypeError`

If the Timestamp is tz-aware and tz is not None.

## `pandas.Timestamp.tzname`

`Timestamp.tzname()`  
Return `self.tzinfo.tzname(self)`.

## `pandas.Timestamp.utctimestamp`

**classmethod** `Timestamp.utctimestamp(ts)`  
Construct a naive UTC datetime from a POSIX timestamp.

## `pandas.Timestamp.utcnow`

**classmethod** `Timestamp.utcnow()`  
Return a new Timestamp representing UTC day and time.

## `pandas.Timestamp.utcoffset`

`Timestamp.utcoffset()`  
Return `self.tzinfo.utcoffset(self)`.

## `pandas.Timestamp.utctimetuple`

`Timestamp.utctimetuple()`  
Return UTC time tuple, compatible with `time.localtime()`.

## `pandas.Timestamp.weekday`

`Timestamp.weekday()`  
Return the day of the week represented by the date. Monday == 0 ... Sunday == 6

isoformat	
-----------	--

### 34.14.5 Properties

<code>Timestamp.asm8</code>	
<code>Timestamp.day</code>	
<code>Timestamp.dayofweek</code>	
<code>Timestamp.dayofyear</code>	
<code>Timestamp.days_in_month</code>	
<code>Timestamp.daysinmonth</code>	
<code>Timestamp.fold</code>	
<code>Timestamp.hour</code>	
<code>Timestamp.is_leap_year</code>	
<code>Timestamp.is_month_end</code>	
<code>Timestamp.is_month_start</code>	
<code>Timestamp.is_quarter_end</code>	
<code>Timestamp.is_quarter_start</code>	
<code>Timestamp.is_year_end</code>	
<code>Timestamp.is_year_start</code>	
<code>Timestamp.max</code>	
<code>Timestamp.microsecond</code>	
<code>Timestamp.min</code>	
<code>Timestamp.minute</code>	
<code>Timestamp.month</code>	
<code>Timestamp.nanosecond</code>	
<code>Timestamp.quarter</code>	
<code>Timestamp.resolution</code>	
<code>Timestamp.second</code>	
<code>Timestamp.tz</code>	Alias for <code>tzinfo</code>
<code>Timestamp.tzinfo</code>	
<code>Timestamp.value</code>	
<code>Timestamp.week</code>	
<code>Timestamp.weekofyear</code>	
<code>Timestamp.year</code>	

#### 34.14.5.1 pandas.Timestamp.asm8

`Timestamp.asm8`

#### 34.14.5.2 pandas.Timestamp.day

`Timestamp.day`

#### 34.14.5.3 pandas.Timestamp.dayofweek

`Timestamp.dayofweek`

#### 34.14.5.4 pandas.Timestamp.dayofyear

`Timestamp.dayofyear`

#### **34.14.5.5 pandas.Timestamp.days\_in\_month**

`Timestamp.days_in_month`

#### **34.14.5.6 pandas.Timestamp.daysinmonth**

`Timestamp.daysinmonth`

#### **34.14.5.7 pandas.Timestamp.fold**

`Timestamp.fold`

#### **34.14.5.8 pandas.Timestamp.hour**

`Timestamp.hour`

#### **34.14.5.9 pandas.Timestamp.is\_leap\_year**

`Timestamp.is_leap_year`

#### **34.14.5.10 pandas.Timestamp.is\_month\_end**

`Timestamp.is_month_end`

#### **34.14.5.11 pandas.Timestamp.is\_month\_start**

`Timestamp.is_month_start`

#### **34.14.5.12 pandas.Timestamp.is\_quarter\_end**

`Timestamp.is_quarter_end`

#### **34.14.5.13 pandas.Timestamp.is\_quarter\_start**

`Timestamp.is_quarter_start`

#### **34.14.5.14 pandas.Timestamp.is\_year\_end**

`Timestamp.is_year_end`

#### **34.14.5.15 pandas.Timestamp.is\_year\_start**

`Timestamp.is_year_start`

#### 34.14.5.16 pandas.Timestamp.max

`Timestamp.max = Timestamp('2262-04-11 23:47:16.854775807')`

#### 34.14.5.17 pandas.Timestamp.microsecond

`Timestamp.microsecond`

#### 34.14.5.18 pandas.Timestamp.min

`Timestamp.min = Timestamp('1677-09-21 00:12:43.145225')`

#### 34.14.5.19 pandas.Timestamp.minute

`Timestamp.minute`

#### 34.14.5.20 pandas.Timestamp.month

`Timestamp.month`

#### 34.14.5.21 pandas.Timestamp.nanosecond

`Timestamp.nanosecond`

#### 34.14.5.22 pandas.Timestamp.quarter

`Timestamp.quarter`

#### 34.14.5.23 pandas.Timestamp.resolution

`Timestamp.resolution = datetime.timedelta(0, 0, 1)`

#### 34.14.5.24 pandas.Timestamp.second

`Timestamp.second`

#### 34.14.5.25 pandas.Timestamp.tzinfo

`Timestamp.tzinfo`

#### 34.14.5.26 pandas.Timestamp.value

`Timestamp.value`

**34.14.5.27 pandas.Timestamp.week**`Timestamp.week`**34.14.5.28 pandas.Timestamp.weekofyear**`Timestamp.weekofyear`**34.14.5.29 pandas.Timestamp.year**`Timestamp.year`**34.14.6 Methods**

<code>Timestamp.astimezone</code>	Convert tz-aware Timestamp to another time zone.
<code>Timestamp.ceil</code>	return a new Timestamp ceiled to this resolution
<code>Timestamp.combine(date, time)</code>	date, time -> datetime with same date and time fields
<code>Timestamp.ctime</code>	Return ctime() style string.
<code>Timestamp.date</code>	Return date object with same year, month and day.
<code>Timestamp.day_name</code>	Return the day name of the Timestamp with specified locale.
<code>Timestamp.dst</code>	Return self.tzinfo.dst(self).
<code>Timestamp.floor</code>	return a new Timestamp floored to this resolution
<code>Timestamp.freq</code>	
<code>Timestamp.freqstr</code>	
<code>Timestamp.fromordinal(ordinal[, freq, tz])</code>	passed an ordinal, translate and convert to a ts note: by definition there cannot be any tz info on the ordinal itself
<code>Timestamp.fromtimestamp(ts)</code>	timestamp[, tz] -> tz's local time from POSIX timestamp.
<code>Timestamp.isocalendar</code>	Return a 3-tuple containing ISO year, week number, and weekday.
<code>Timestamp.isoformat</code>	
<code>Timestamp.isoweekday</code>	Return the day of the week represented by the date.
<code>Timestamp.month_name</code>	Return the month name of the Timestamp with specified locale.
<code>Timestamp.normalize</code>	Normalize Timestamp to midnight, preserving tz information.
<code>Timestamp.now([tz])</code>	Returns new Timestamp object representing current time local to tz.
<code>Timestamp.replace</code>	implements datetime.replace, handles nanoseconds
<code>Timestamp.round</code>	Round the Timestamp to the specified resolution
<code>Timestamp.strftime</code>	format -> strftime() style string.
<code>Timestamp.strptime</code>	string, format -> new datetime parsed from a string (like time.strptime()).
<code>Timestamp.time</code>	Return time object with same time but with tz-info=None.
<code>Timestamp.timestamp</code>	Return POSIX timestamp as float.
<code>Timestamp.timetuple</code>	Return time tuple, compatible with time.localtime().
<code>Timestamp.timetz</code>	Return time object with same time and tzinfo.

Continued on next page



Table 150 – continued from previous page

<i>Timestamp.to_datetime64</i>	Returns a <code>numpy.datetime64</code> object with ‘ns’ precision
<i>Timestamp.to_julian_date</i>	Convert <code>TimeStamp</code> to a Julian Date.
<i>Timestamp.to_period</i>	Return an period of which this timestamp is an observation.
<i>Timestamp.to_pydatetime</i>	Convert a <code>TimeStamp</code> object to a native Python <code>datetime</code> object.
<i>Timestamp.today(cls[, tz])</i>	Return the current time in the local timezone.
<i>Timestamp.toordinal</i>	Return proleptic Gregorian ordinal.
<i>Timestamp.tz_convert</i>	Convert tz-aware <code>TimeStamp</code> to another time zone.
<i>Timestamp.tz_localize</i>	Convert naive <code>TimeStamp</code> to local time zone, or remove timezone from tz-aware <code>TimeStamp</code> .
<i>Timestamp.tzname</i>	Return <code>self.tzinfo.tzname(self)</code> .
<i>Timestamp.utctimestamp(ts)</i>	Construct a naive UTC <code>datetime</code> from a POSIX timestamp.
<i>Timestamp.utcnow()</i>	Return a new <code>TimeStamp</code> representing UTC day and time.
<i>Timestamp.utcoffset</i>	Return <code>self.tzinfo.utcoffset(self)</code> .
<i>Timestamp.utctimetuple</i>	Return UTC time tuple, compatible with <code>time.localtime()</code> .
<i>Timestamp.weekday</i>	Return the day of the week represented by the date.

#### 34.14.6.1 pandas.Timestamp.freq

`Timestamp.freq`

#### 34.14.6.2 pandas.Timestamp.freqstr

`Timestamp.freqstr`

#### 34.14.6.3 pandas.Timestamp.isoformat

`Timestamp.isoformat`

### 34.14.7 Interval

<i>Interval</i>	Immutable object implementing an Interval, a bounded slice-like interval.
-----------------	---

#### 34.14.7.1 pandas.Interval

**class** `pandas.Interval`

Immutable object implementing an Interval, a bounded slice-like interval.

New in version 0.20.0.

**Parameters** `left` : orderable scalar

Left bound for the interval.

**right** : orderable scalar

Right bound for the interval.

**closed** : { 'left', 'right', 'both', 'neither' }, default 'right'

Whether the interval is closed on the left-side, right-side, both or neither.

**closed** : { 'right', 'left', 'both', 'neither' }, default 'right'

Whether the interval is closed on the left-side, right-side, both or neither. See the Notes for more detailed explanation.

**See also:**

**IntervalIndex** An Index of Interval objects that are all closed on the same side.

**cut** Convert continuous data into discrete bins (Categorical of Interval objects).

**qcut** Convert continuous data into bins (Categorical of Interval objects) based on quantiles.

**Period** Represents a period of time.

## Notes

The parameters *left* and *right* must be from the same type, you must be able to compare them and they must satisfy `left <= right`.

A closed interval (in mathematics denoted by square brackets) contains its endpoints, i.e. the closed interval `[0, 5]` is characterized by the conditions `0 <= x <= 5`. This is what `closed='both'` stands for. An open interval (in mathematics denoted by parentheses) does not contain its endpoints, i.e. the open interval `(0, 5)` is characterized by the conditions `0 < x < 5`. This is what `closed='neither'` stands for. Intervals can also be half-open or half-closed, i.e. `[0, 5)` is described by `0 <= x < 5` (`closed='left'`) and `(0, 5]` is described by `0 < x <= 5` (`closed='right'`).

## Examples

It is possible to build Intervals of different types, like numeric ones:

```
>>> iv = pd.Interval(left=0, right=5)
>>> iv
Interval(0, 5, closed='right')
```

You can check if an element belongs to it

```
>>> 2.5 in iv
True
```

You can test the bounds (`closed='right'`, so `0 < x <= 5`):

```
>>> 0 in iv
False
>>> 5 in iv
True
>>> 0.0001 in iv
True
```

Calculate its length

```
>>> iv.length
5
```

You can operate with + and \* over an Interval and the operation is applied to each of its bounds, so the result depends on the type of the bound elements

```
>>> shifted_iv = iv + 3
>>> shifted_iv
Interval(3, 8, closed='right')
>>> extended_iv = iv * 10.0
>>> extended_iv
Interval(0.0, 50.0, closed='right')
```

To create a time interval you can use Timestamps as the bounds

```
>>> year_2017 = pd.Interval(pd.Timestamp('2017-01-01 00:00:00'),
...                          pd.Timestamp('2018-01-01 00:00:00'),
...                          closed='left')
>>> pd.Timestamp('2017-01-01 00:00:00') in year_2017
True
>>> year_2017.length
Timedelta('365 days 00:00:00')
```

And also you can create string intervals

```
>>> volume_1 = pd.Interval('Ant', 'Dog', closed='both')
>>> 'Bee' in volume_1
True
```

## Attributes

<i>closed</i>	Whether the interval is closed on the left-side, right-side, both or neither
<i>closed_left</i>	Check if the interval is closed on the left side.
<i>closed_right</i>	Check if the interval is closed on the right side.
<i>left</i>	Left bound for the interval
<i>length</i>	Return the length of the Interval
<i>mid</i>	Return the midpoint of the Interval
<i>open_left</i>	Check if the interval is open on the left side.
<i>open_right</i>	Check if the interval is open on the right side.
<i>right</i>	Right bound for the interval

### pandas.Interval.closed

`Interval.closed`

Whether the interval is closed on the left-side, right-side, both or neither

### pandas.Interval.closed\_left

`Interval.closed_left`

Check if the interval is closed on the left side.

For the meaning of *closed* and *open* see [Interval](#).

**Returns bool**

True if the Interval is closed on the left-side, else False.

**pandas.Interval.closed\_right**

`Interval.closed_right`

Check if the interval is closed on the right side.

For the meaning of *closed* and *open* see [Interval](#).

**Returns bool**

True if the Interval is closed on the left-side, else False.

**pandas.Interval.left**

`Interval.left`

Left bound for the interval

**pandas.Interval.length**

`Interval.length`

Return the length of the Interval

**pandas.Interval.mid**

`Interval.mid`

Return the midpoint of the Interval

**pandas.Interval.open\_left**

`Interval.open_left`

Check if the interval is open on the left side.

For the meaning of *closed* and *open* see [Interval](#).

**Returns bool**

True if the Interval is closed on the left-side, else False.

**pandas.Interval.open\_right**

`Interval.open_right`

Check if the interval is open on the right side.

For the meaning of *closed* and *open* see [Interval](#).

**Returns bool**

True if the Interval is closed on the left-side, else False.

**pandas.Interval.right**`Interval.right`

Right bound for the interval

**34.14.8 Properties**

<code>Interval.closed</code>	Whether the interval is closed on the left-side, right-side, both or neither
<code>Interval.closed_left</code>	Check if the interval is closed on the left side.
<code>Interval.closed_right</code>	Check if the interval is closed on the right side.
<code>Interval.left</code>	Left bound for the interval
<code>Interval.length</code>	Return the length of the Interval
<code>Interval.mid</code>	Return the midpoint of the Interval
<code>Interval.open_left</code>	Check if the interval is open on the left side.
<code>Interval.open_right</code>	Check if the interval is open on the right side.
<code>Interval.right</code>	Right bound for the interval

**34.14.9 Timedelta**

<code>Timedelta</code>	Represents a duration, the difference between two dates or times.
------------------------	---

**34.14.9.1 pandas.Timedelta****class** `pandas.Timedelta`

Represents a duration, the difference between two dates or times.

Timedelta is the pandas equivalent of python's `datetime.timedelta` and is interchangeable with it in most cases.

**Parameters****value** [Timedelta, timedelta, np.timedelta64, string, or integer]**unit** : string, {'ns', 'us', 'ms', 's', 'm', 'h', 'D'}, optional

Denote the unit of the input, if input is an integer. Default 'ns'.

**days, seconds, microseconds,****milliseconds, minutes, hours, weeks** : numeric, optional

Values for construction in compat with `datetime.timedelta`. np ints and floats will be coerced to python ints and floats.

**Notes**

The `.value` attribute is always in ns.

## Attributes

<code>asm8</code>	return a numpy timedelta64 array view of myself
<code>components</code>	Return a Components NamedTuple-like
<code>days</code>	Number of days.
<code>delta</code>	Return the timedelta in nanoseconds (ns), for internal compatibility.
<code>microseconds</code>	Number of microseconds ( $\geq 0$ and less than 1 second).
<code>nanoseconds</code>	Return the number of nanoseconds (n), where $0 \leq n < 1$ microsecond.
<code>resolution</code>	return a string representing the lowest resolution that we have
<code>seconds</code>	Number of seconds ( $\geq 0$ and less than 1 day).

### `pandas.Timedelta.asm8`

`Timedelta.asm8`  
return a numpy timedelta64 array view of myself

### `pandas.Timedelta.components`

`Timedelta.components`  
Return a Components NamedTuple-like

### `pandas.Timedelta.days`

`Timedelta.days`  
Number of days.

### `pandas.Timedelta.delta`

`Timedelta.delta`  
Return the timedelta in nanoseconds (ns), for internal compatibility.

#### Returns `int`

Timedelta in nanoseconds.

## Examples

```
>>> td = pd.Timedelta('1 days 42 ns')
>>> td.delta
86400000000042
```

```
>>> td = pd.Timedelta('3 s')
>>> td.delta
3000000000
```

```
>>> td = pd.Timedelta('3 ms 5 us')
>>> td.delta
3005000
```

```
>>> td = pd.Timedelta(42, unit='ns')
>>> td.delta
42
```

### pandas.Timedelta.microseconds

`Timedelta.microseconds`

Number of microseconds ( $\geq 0$  and less than 1 second).

### pandas.Timedelta.nanoseconds

`Timedelta.nanoseconds`

Return the number of nanoseconds (n), where  $0 \leq n < 1$  microsecond.

**Returns** int

Number of nanoseconds.

**See also:**

***Timedelta.components*** Return all attributes with assigned values (i.e. days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds).

### Examples

#### Using string input

```
>>> td = pd.Timedelta('1 days 2 min 3 us 42 ns')
>>> td.nanoseconds
42
```

#### Using integer input

```
>>> td = pd.Timedelta(42, unit='ns')
>>> td.nanoseconds
42
```

### pandas.Timedelta.resolution

`Timedelta.resolution`

return a string representing the lowest resolution that we have

### pandas.Timedelta.seconds

`Timedelta.seconds`

Number of seconds ( $\geq 0$  and less than 1 day).

<b>freq</b>	
<b>is_populated</b>	
<b>value</b>	

## Methods

<i>ceil</i>	return a new Timedelta ceiled to this resolution
<i>floor</i>	return a new Timedelta floored to this resolution
<i>isoformat</i>	Format Timedelta as ISO 8601 Duration like P[n]Y[n]M[n]DT[n]H[n]M[n]S, where the [n] s are replaced by the values.
<i>round</i>	Round the Timedelta to the specified resolution
<i>to_pytimedelta</i>	return an actual datetime.timedelta object note: we lose nanosecond resolution if any
<i>to_timedelta64</i>	Returns a numpy.timedelta64 object with 'ns' precision
<i>total_seconds</i>	Total duration of timedelta in seconds (to ns precision)
<i>view</i>	array view compat

### pandas.Timedelta.ceil

`Timedelta.ceil`

return a new Timedelta ceiled to this resolution

#### Parameters

**freq** [a freq string indicating the ceiling resolution]

### pandas.Timedelta.floor

`Timedelta.floor`

return a new Timedelta floored to this resolution

#### Parameters

**freq** [a freq string indicating the flooring resolution]

### pandas.Timedelta.isoformat

`Timedelta.isoformat()`

Format Timedelta as ISO 8601 Duration like P[n]Y[n]M[n]DT[n]H[n]M[n]S, where the [n] s are replaced by the values. See [https://en.wikipedia.org/wiki/ISO\\_8601#Durations](https://en.wikipedia.org/wiki/ISO_8601#Durations)

New in version 0.20.0.

#### Returns

**formatted** [str]

**See also:**

*Timestamp.isoformat*



## Notes

The longest component is days, whose value may be larger than 365. Every component is always included, even if its value is 0. Pandas uses nanosecond precision, so up to 9 decimal places may be included in the seconds component. Trailing 0's are removed from the seconds component after the decimal. We do not 0 pad components, so it's ... *T5H...*, not ... *T05H...*

## Examples

```
>>> td = pd.Timedelta(days=6, minutes=50, seconds=3,
...                   milliseconds=10, microseconds=10, nanoseconds=12)
>>> td.isoformat()
'P6DT0H50M3.010010012S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT0H0M10S'
>>> pd.Timedelta(hours=1, seconds=10).isoformat()
'P0DT0H0M10S'
>>> pd.Timedelta(days=500.5).isoformat()
'P500DT12H0MS'
```

## pandas.Timedelta.round

`Timedelta.round`

Round the Timedelta to the specified resolution

### Parameters

**freq** [a freq string indicating the rounding resolution]

### Returns

a new **Timedelta** rounded to the given resolution of 'freq'

### Raises

**ValueError** if the freq cannot be converted

## pandas.Timedelta.to\_pytimedelta

`Timedelta.to_pytimedelta()`

return an actual datetime.timedelta object note: we lose nanosecond resolution if any

## pandas.Timedelta.to\_timedelta64

`Timedelta.to_timedelta64()`

Returns a numpy.timedelta64 object with 'ns' precision

## pandas.Timedelta.total\_seconds

`Timedelta.total_seconds()`

Total duration of timedelta in seconds (to ns precision)

**pandas.Timedelta.view**

```
Timedelta.view()  
array view compat
```

**34.14.10 Properties**

<i>Timedelta.asm8</i>	return a numpy timedelta64 array view of myself
<i>Timedelta.components</i>	Return a Components NamedTuple-like
<i>Timedelta.days</i>	Number of days.
<i>Timedelta.delta</i>	Return the timedelta in nanoseconds (ns), for internal compatibility.
<i>Timedelta.freq</i>	
<i>Timedelta.is_populated</i>	
<i>Timedelta.max</i>	
<i>Timedelta.microseconds</i>	Number of microseconds ( $\geq 0$ and less than 1 second).
<i>Timedelta.min</i>	
<i>Timedelta.nanoseconds</i>	Return the number of nanoseconds (n), where $0 \leq n < 1$ microsecond.
<i>Timedelta.resolution</i>	return a string representing the lowest resolution that we have
<i>Timedelta.seconds</i>	Number of seconds ( $\geq 0$ and less than 1 day).
<i>Timedelta.value</i>	
<i>Timedelta.view</i>	array view compat

**34.14.10.1 pandas.Timedelta.freq**

```
Timedelta.freq
```

**34.14.10.2 pandas.Timedelta.is\_populated**

```
Timedelta.is_populated
```

**34.14.10.3 pandas.Timedelta.max**

```
Timedelta.max = Timedelta('106751 days 23:47:16.854775')
```

**34.14.10.4 pandas.Timedelta.min**

```
Timedelta.min = Timedelta('-106752 days +00:12:43.145224')
```

**34.14.10.5 pandas.Timedelta.value**

```
Timedelta.value
```

**34.14.11 Methods**

<code>Timedelta.ceil</code>	return a new Timedelta ceiled to this resolution
<code>Timedelta.floor</code>	return a new Timedelta floored to this resolution
<code>Timedelta.isoformat</code>	Format Timedelta as ISO 8601 Duration like P[n]Y[n]M[n]DT[n]H[n]M[n]S, where the [n] s are replaced by the values.
<code>Timedelta.round</code>	Round the Timedelta to the specified resolution
<code>Timedelta.to_pytimedelta</code>	return an actual datetime.timedelta object note: we lose nanosecond resolution if any
<code>Timedelta.to_timedelta64</code>	Returns a numpy.timedelta64 object with 'ns' precision
<code>Timedelta.total_seconds</code>	Total duration of timedelta in seconds (to ns precision)

## 34.15 Frequencies

<code>to_offset(freq)</code>	Return DateOffset object from string or tuple representation or datetime.timedelta object
------------------------------	---

### 34.15.1 pandas.tseries.frequencies.to\_offset

`pandas.tseries.frequencies.to_offset(freq)`

Return DateOffset object from string or tuple representation or datetime.timedelta object

#### Parameters

**freq** [str, tuple, datetime.timedelta, DateOffset or None]

**Returns** `delta` : DateOffset

None if freq is None

**Raises** `ValueError`

If freq is an invalid frequency

**See also:**

`pandas.DateOffset`

#### Examples

```
>>> to_offset('5min')
<5 * Minutes>
```

```
>>> to_offset('1D1H')
<25 * Hours>
```

```
>>> to_offset(('W', 2))
<2 * Weeks: weekday=6>
```

```
>>> to_offset((2, 'B'))
<2 * BusinessDays>
```

```
>>> to_offset(datetime.timedelta(days=1))
<Day>
```

```
>>> to_offset(Hour())
<Hour>
```

## 34.16 Window

Rolling objects are returned by `.rolling` calls: `pandas.DataFrame.rolling()`, `pandas.Series.rolling()`, etc. Expanding objects are returned by `.expanding` calls: `pandas.DataFrame.expanding()`, `pandas.Series.expanding()`, etc. EWM objects are returned by `.ewm` calls: `pandas.DataFrame.ewm()`, `pandas.Series.ewm()`, etc.

### 34.16.1 Standard moving window functions

<code>Rolling.count()</code>	The rolling count of any non-NaN observations inside the window.
<code>Rolling.sum(*args, **kwargs)</code>	Calculate rolling sum of given DataFrame or Series.
<code>Rolling.mean(*args, **kwargs)</code>	Calculate the rolling mean of the values.
<code>Rolling.median(**kwargs)</code>	Calculate the rolling median.
<code>Rolling.var([ddof])</code>	Calculate unbiased rolling variance.
<code>Rolling.std([ddof])</code>	Calculate rolling standard deviation.
<code>Rolling.min(*args, **kwargs)</code>	Calculate the rolling minimum.
<code>Rolling.max(*args, **kwargs)</code>	rolling maximum
<code>Rolling.corr([other, pairwise])</code>	rolling sample correlation
<code>Rolling.cov([other, pairwise, ddof])</code>	rolling sample covariance
<code>Rolling.skew(**kwargs)</code>	Unbiased rolling skewness
<code>Rolling.kurt(**kwargs)</code>	Calculate unbiased rolling kurtosis.
<code>Rolling.apply(func[, raw, args, kwargs])</code>	rolling function apply
<code>Rolling.aggregate(arg, *args, **kwargs)</code>	Aggregate using one or more operations over the specified axis.
<code>Rolling.quantile(quantile[, interpolation])</code>	rolling quantile.
<code>Window.mean(*args, **kwargs)</code>	Calculate the window mean of the values.
<code>Window.sum(*args, **kwargs)</code>	Calculate window sum of given DataFrame or Series.

#### 34.16.1.1 pandas.core.window.Rolling.count

`Rolling.count()`

The rolling count of any non-NaN observations inside the window.

**Returns Series or DataFrame**

Returned object type is determined by the caller of the rolling calculation.

**See also:**

`pandas.Series.rolling` Calling object with Series data

`pandas.DataFrame.rolling` Calling object with DataFrames

`pandas.DataFrame.count` Count of the full DataFrame

## Examples

```

>>> s = pd.Series([2, 3, np.nan, 10])
>>> s.rolling(2).count()
0    1.0
1    2.0
2    1.0
3    1.0
dtype: float64
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.rolling(4).count()
0    1.0
1    2.0
2    2.0
3    3.0
dtype: float64

```

### 34.16.1.2 pandas.core.window.Rolling.sum

Rolling.**sum**(\*args, \*\*kwargs)

Calculate rolling sum of given DataFrame or Series.

**Parameters** \*args, \*\*kwargs

For compatibility with other rolling methods. Has no effect on the computed value.

**Returns** Series or DataFrame

Same type as the input, with the same index, containing the rolling sum.

**See also:**

**Series.sum** Reducing sum for Series.

**DataFrame.sum** Reducing sum for DataFrame.

## Examples

```

>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64

```

```

>>> s.rolling(3).sum()
0    NaN
1    NaN

```

(continues on next page)

(continued from previous page)

```

2      6.0
3      9.0
4     12.0
dtype: float64

```

```

>>> s.expanding(3).sum()
0      NaN
1      NaN
2      6.0
3     10.0
4     15.0
dtype: float64

```

```

>>> s.rolling(3, center=True).sum()
0      NaN
1      6.0
2      9.0
3     12.0
4      NaN
dtype: float64

```

For DataFrame, each rolling sum is computed column-wise.

```

>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A  B
0  1  1
1  2  4
2  3  9
3  4 16
4  5 25

```

```

>>> df.rolling(3).sum()
   A      B
0  NaN  NaN
1  NaN  NaN
2  6.0 14.0
3  9.0 29.0
4 12.0 50.0

```

### 34.16.1.3 pandas.core.window.Rolling.mean

Rolling.**mean**(\*args, \*\*kwargs)

Calculate the rolling mean of the values.

#### Parameters **\*args**

Under Review.

#### **\*\*kwargs**

Under Review.

#### Returns **Series or DataFrame**

Returned object type is determined by the caller of the rolling calculation.

See also:

**Series.rolling** Calling object with Series data

**DataFrame.rolling** Calling object with DataFrames

**Series.mean** Equivalent method for Series

**DataFrame.mean** Equivalent method for DataFrame

## Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0      NaN
1      1.5
2      2.5
3      3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0      NaN
1      NaN
2      2.0
3      3.0
dtype: float64
```

### 34.16.1.4 pandas.core.window.Rolling.median

**Rolling.median** (\*\*kwargs)

Calculate the rolling median.

**Parameters** \*\*kwargs

For compatibility with other rolling methods. Has no effect on the computed median.

**Returns** Series or DataFrame

Returned type is the same as the original object.

See also:

**Series.rolling** Calling object with Series data

**DataFrame.rolling** Calling object with DataFrames

**Series.median** Equivalent method for Series

**DataFrame.median** Equivalent method for DataFrame

## Examples

Compute the rolling median of a series with a window size of 3.

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.rolling(3).median()
0      NaN
1      NaN
2      1.0
3      2.0
4      3.0
dtype: float64
```

### 34.16.1.5 pandas.core.window.Rolling.var

`Rolling.var(ddof=1, *args, **kwargs)`

Calculate unbiased rolling variance.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

**Parameters** *ddof*: int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

*\*args, \*\*kwargs*

For NumPy compatibility. No additional arguments are used.

**Returns** **Series or DataFrame**

Returns the same object type as the caller of the rolling calculation.

**See also:**

**Series.rolling** Calling object with Series data

**DataFrame.rolling** Calling object with DataFrames

**Series.var** Equivalent method for Series

**DataFrame.var** Equivalent method for DataFrame

**numpy.var** Equivalent method for Numpy array

### Notes

The default *ddof* of 1 used in `Series.var()` is different than the default *ddof* of 0 in `numpy.var()`.

A minimum of 1 period is required for the rolling calculation.

### Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0      NaN
1      NaN
2    0.333333
3    1.000000
4    1.000000
5    1.333333
```

(continues on next page)



(continued from previous page)

```
6      0.000000
dtype: float64
```

```
>>> s.expanding(3).var()
0      NaN
1      NaN
2      0.333333
3      0.916667
4      0.800000
5      0.700000
6      0.619048
dtype: float64
```

### 34.16.1.6 pandas.core.window.Rolling.std

`Rolling.std(ddof=1, *args, **kwargs)`

Calculate rolling standard deviation.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

**Parameters** *ddof* : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**\*args, \*\*kwargs**

For NumPy compatibility. No additional arguments are used.

**Returns** **Series or DataFrame**

Returns the same object type as the caller of the rolling calculation.

**See also:**

**Series.rolling** Calling object with Series data

**DataFrame.rolling** Calling object with DataFrames

**Series.std** Equivalent method for Series

**DataFrame.std** Equivalent method for DataFrame

**numpy.std** Equivalent method for Numpy array

### Notes

The default *ddof* of 1 used in `Series.std` is different than the default *ddof* of 0 in `numpy.std`.

A minimum of one period is required for the rolling calculation.

### Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0      NaN
1      NaN
2    0.577350
3    1.000000
4    1.000000
5    1.154701
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).std()
0      NaN
1      NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

### 34.16.1.7 pandas.core.window.Rolling.min

`Rolling.min(*args, **kwargs)`  
Calculate the rolling minimum.

**Parameters** **\*\*kwargs**

Under Review.

**Returns** **Series or DataFrame**

Returned object type is determined by the caller of the rolling calculation.

**See also:**

**Series.rolling** Calling object with a Series

**DataFrame.rolling** Calling object with a DataFrame

**Series.min** Similar method for Series

**DataFrame.min** Similar method for DataFrame

### Examples

Performing a rolling minimum with a window size of 3.

```
>>> s = pd.Series([4, 3, 5, 2, 6])
>>> s.rolling(3).min()
0      NaN
1      NaN
2     3.0
3     2.0
4     2.0
dtype: float64
```

### 34.16.1.8 pandas.core.window.Rolling.max

`Rolling.max(*args, **kwargs)`  
 rolling maximum

**Returns**

same type as input

**See also:**

*pandas.Series.rolling, pandas.DataFrame.rolling*

### 34.16.1.9 pandas.core.window.Rolling.corr

`Rolling.corr(other=None, pairwise=None, **kwargs)`  
 rolling sample correlation

**Parameters** **other** : Series, DataFrame, or ndarray, optional

if not supplied then will default to self and produce pairwise output

**pairwise** : bool, default None

If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**Returns**

same type as input

**See also:**

*pandas.Series.rolling, pandas.DataFrame.rolling*

### 34.16.1.10 pandas.core.window.Rolling.cov

`Rolling.cov(other=None, pairwise=None, ddof=1, **kwargs)`  
 rolling sample covariance

**Parameters** **other** : Series, DataFrame, or ndarray, optional

if not supplied then will default to self and produce pairwise output

**pairwise** : bool, default None

If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**Returns**

same type as input

See also:

*pandas.Series.rolling, pandas.DataFrame.rolling*

#### 34.16.1.11 pandas.core.window.Rolling.skew

`Rolling.skew(**kwargs)`

Unbiased rolling skewness

**Returns**

same type as input

See also:

*pandas.Series.rolling, pandas.DataFrame.rolling*

#### 34.16.1.12 pandas.core.window.Rolling.kurt

`Rolling.kurt(**kwargs)`

Calculate unbiased rolling kurtosis.

This function uses Fisher's definition of kurtosis without bias.

**Parameters** `**kwargs`

Under Review.

**Returns** **Series or DataFrame**

Returned object type is determined by the caller of the rolling calculation

See also:

**Series.rolling** Calling object with Series data

**DataFrame.rolling** Calling object with DataFrames

**Series.kurt** Equivalent method for Series

**DataFrame.kurt** Equivalent method for DataFrame

**scipy.stats.skew** Third moment of a probability density

**scipy.stats.kurtosis** Reference SciPy method

#### Notes

A minimum of 4 periods is required for the rolling calculation.

#### Examples

The example below will show a rolling calculation with a window size of four matching the equivalent function call using *scipy.stats*.

```

>>> arr = [1, 2, 3, 4, 999]
>>> fmt = "{0:.6f}" # limit the printed precision to 6 digits
>>> import scipy.stats
>>> print(fmt.format(scipy.stats.kurtosis(arr[:-1], bias=False)))
-1.200000
>>> print(fmt.format(scipy.stats.kurtosis(arr[1:], bias=False)))
3.999946
>>> s = pd.Series(arr)
>>> s.rolling(4).kurt()
0      NaN
1      NaN
2      NaN
3    -1.200000
4     3.999946
dtype: float64

```

### 34.16.1.13 pandas.core.window.Rolling.apply

`Rolling.apply` (*func*, *raw=None*, *args=()*, *kwargs={}*)  
 rolling function apply

**Parameters** *func* : function

Must produce a single value from an ndarray input if *raw=True* or a Series if *raw=False*

**raw** : bool, default None

- *False* : passes each row or column as a Series to the function.
- *True* or *None* : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

The *raw* parameter is required and will show a FutureWarning if not passed. In the future *raw* will default to *False*.

New in version 0.23.0.

**\*args and \*\*kwargs are passed to the function**

**Returns**

same type as input

**See also:**

`pandas.Series.rolling`, `pandas.DataFrame.rolling`

### 34.16.1.14 pandas.core.window.Rolling.aggregate

`Rolling.aggregate` (*arg*, *\*args*, *\*\*kwargs*)

Aggregate using one or more operations over the specified axis.

**Parameters** *func* : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a Series/DataFrame or when passed to Series/DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.

**Returns**

**aggregated** [Series/DataFrame]

**See also:**

`pandas.Series.rolling`, `pandas.DataFrame.rolling`

**Notes**

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

**Examples**

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'])
>>> df
```

	A	B	C
0	-2.385977	-0.102758	0.438822
1	-1.004295	0.905829	-0.954544
2	0.735167	-0.165272	-1.619346
3	-0.702657	-1.340923	-0.706334
4	-0.246845	0.211596	-0.901819
5	2.463718	3.157577	-1.380906
6	-1.142255	2.340594	-0.039875
7	1.396598	-1.647453	1.677227
8	-0.543425	1.761277	-0.220481
9	-0.640505	0.289374	-1.550670

```
>>> df.rolling(3).sum()
```

	A	B	C
0	NaN	NaN	NaN
1	NaN	NaN	NaN
2	-2.655105	0.637799	-2.135068
3	-0.971785	-0.600366	-3.280224
4	-0.214334	-1.294599	-3.227500
5	1.514216	2.028250	-2.989060
6	1.074618	5.709767	-2.322600
7	2.718061	3.850718	0.256446
8	-0.289082	2.454418	1.416871
9	0.212668	0.403198	-0.093924

```
>>> df.rolling(3).agg({'A': 'sum', 'B': 'min'})
```

	A	B
0	NaN	NaN
1	NaN	NaN
2	-2.655105	-0.165272
3	-0.971785	-1.340923
4	-0.214334	-1.340923
5	1.514216	-1.340923
6	1.074618	0.211596
7	2.718061	-1.647453
8	-0.289082	-1.647453
9	0.212668	-1.647453

### 34.16.1.15 pandas.core.window.Rolling.quantile

`Rolling.quantile` (*quantile*, *interpolation*='linear', *\*\*kwargs*)  
rolling quantile.

**Parameters** *quantile* : float

Quantile to compute.  $0 \leq \text{quantile} \leq 1$ .

**interpolation** : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}

New in version 0.23.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear:  $i + (j - i) * \text{fraction}$ , where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint:  $(i + j) / 2$ .

**\*\*kwargs**:

For compatibility with other rolling methods. Has no effect on the result.

**Returns** Series or DataFrame

Returned object type is determined by the caller of the rolling calculation.

See also:

[`pandas.Series.quantile`](#) Computes value at the given quantile over all data in Series.

[`pandas.DataFrame.quantile`](#) Computes values at the given quantile over requested axis in DataFrame.

### Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
```

0	NaN
---	-----

(continues on next page)

(continued from previous page)

```
1    1.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

### 34.16.1.16 pandas.core.window.Window.mean

Window.**mean** (\*args, \*\*kwargs)

Calculate the window mean of the values.

**Parameters** \*args

Under Review.

**\*\*kwargs**

Under Review.

**Returns** Series or DataFrame

Returned object type is determined by the caller of the window calculation.

**See also:**

**Series.window** Calling object with Series data

**DataFrame.window** Calling object with DataFrames

**Series.mean** Equivalent method for Series

**DataFrame.mean** Equivalent method for DataFrame

### Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0    NaN
1    1.5
2    2.5
3    3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0    NaN
1    NaN
2    2.0
3    3.0
dtype: float64
```



### 34.16.1.17 pandas.core.window.Window.sum

`Window.sum(*args, **kwargs)`

Calculate window sum of given DataFrame or Series.

**Parameters** `*args, **kwargs`

For compatibility with other window methods. Has no effect on the computed value.

**Returns** `Series or DataFrame`

Same type as the input, with the same index, containing the window sum.

**See also:**

**Series.sum** Reducing sum for Series.

**DataFrame.sum** Reducing sum for DataFrame.

#### Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0    NaN
1    NaN
2    6.0
3    9.0
4   12.0
dtype: float64
```

```
>>> s.expanding(3).sum()
0    NaN
1    NaN
2    6.0
3   10.0
4   15.0
dtype: float64
```

```
>>> s.rolling(3, center=True).sum()
0    NaN
1    6.0
2    9.0
3   12.0
4    NaN
dtype: float64
```

For DataFrame, each window sum is computed column-wise.

```
>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A  B
0  1  1
1  2  4
2  3  9
3  4 16
4  5 25
```

```
>>> df.rolling(3).sum()
   A    B
0 NaN NaN
1 NaN NaN
2 6.0 14.0
3 9.0 29.0
4 12.0 50.0
```

### 34.16.2 Standard expanding window functions

<i>Expanding.count</i> (**kwargs)	The expanding count of any non-NaN observations inside the window.
<i>Expanding.sum</i> (*args, **kwargs)	Calculate expanding sum of given DataFrame or Series.
<i>Expanding.mean</i> (*args, **kwargs)	Calculate the expanding mean of the values.
<i>Expanding.median</i> (**kwargs)	Calculate the expanding median.
<i>Expanding.var</i> ([ddof])	Calculate unbiased expanding variance.
<i>Expanding.std</i> ([ddof])	Calculate expanding standard deviation.
<i>Expanding.min</i> (*args, **kwargs)	Calculate the expanding minimum.
<i>Expanding.max</i> (*args, **kwargs)	expanding maximum
<i>Expanding.corr</i> ([other, pairwise])	expanding sample correlation
<i>Expanding.cov</i> ([other, pairwise, ddof])	expanding sample covariance
<i>Expanding.skew</i> (**kwargs)	Unbiased expanding skewness
<i>Expanding.kurt</i> (**kwargs)	Calculate unbiased expanding kurtosis.
<i>Expanding.apply</i> (func[, raw, args, kwargs])	expanding function apply
<i>Expanding.aggregate</i> (arg, *args, **kwargs)	Aggregate using one or more operations over the specified axis.
<i>Expanding.quantile</i> (quantile[, interpolation])	expanding quantile.

#### 34.16.2.1 pandas.core.window.Expanding.count

`Expanding.count` (\*\*kwargs)

The expanding count of any non-NaN observations inside the window.

**Returns Series or DataFrame**

Returned object type is determined by the caller of the expanding calculation.

**See also:**

**pandas.Series.expanding** Calling object with Series data

**pandas.DataFrame.expanding** Calling object with DataFrames

**pandas.DataFrame.count** Count of the full DataFrame

## Examples

```
>>> s = pd.Series([2, 3, np.nan, 10])
>>> s.rolling(2).count()
0    1.0
1    2.0
2    1.0
3    1.0
dtype: float64
>>> s.rolling(3).count()
0    1.0
1    2.0
2    2.0
3    2.0
dtype: float64
>>> s.rolling(4).count()
0    1.0
1    2.0
2    2.0
3    3.0
dtype: float64
```

### 34.16.2.2 pandas.core.window.Expanding.sum

Expanding.**sum**(\*args, \*\*kwargs)

Calculate expanding sum of given DataFrame or Series.

**Parameters** \*args, \*\*kwargs

For compatibility with other expanding methods. Has no effect on the computed value.

**Returns** Series or DataFrame

Same type as the input, with the same index, containing the expanding sum.

**See also:**

**Series.sum** Reducing sum for Series.

**DataFrame.sum** Reducing sum for DataFrame.

## Examples

```
>>> s = pd.Series([1, 2, 3, 4, 5])
>>> s
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

```
>>> s.rolling(3).sum()
0    NaN
```

(continues on next page)

(continued from previous page)

```

1      NaN
2      6.0
3      9.0
4     12.0
dtype: float64

```

```

>>> s.expanding(3).sum()
0      NaN
1      NaN
2      6.0
3     10.0
4     15.0
dtype: float64

```

```

>>> s.rolling(3, center=True).sum()
0      NaN
1      6.0
2      9.0
3     12.0
4      NaN
dtype: float64

```

For DataFrame, each expanding sum is computed column-wise.

```

>>> df = pd.DataFrame({"A": s, "B": s ** 2})
>>> df
   A  B
0  1  1
1  2  4
2  3  9
3  4 16
4  5 25

```

```

>>> df.rolling(3).sum()
   A  B
0  NaN NaN
1  NaN NaN
2  6.0 14.0
3  9.0 29.0
4 12.0 50.0

```

### 34.16.2.3 pandas.core.window.Expanding.mean

`Expanding.mean(*args, **kwargs)`

Calculate the expanding mean of the values.

#### Parameters *\*args*

Under Review.

#### *\*\*kwargs*

Under Review.

#### Returns Series or DataFrame

Returned object type is determined by the caller of the expanding calculation.

See also:

**Series.expanding** Calling object with Series data

**DataFrame.expanding** Calling object with DataFrames

**Series.mean** Equivalent method for Series

**DataFrame.mean** Equivalent method for DataFrame

## Examples

The below examples will show rolling mean calculations with window sizes of two and three, respectively.

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).mean()
0      NaN
1      1.5
2      2.5
3      3.5
dtype: float64
```

```
>>> s.rolling(3).mean()
0      NaN
1      NaN
2      2.0
3      3.0
dtype: float64
```

### 34.16.2.4 pandas.core.window.Expanding.median

**Expanding.median** (\*\*kwargs)

Calculate the expanding median.

**Parameters** \*\*kwargs

For compatibility with other expanding methods. Has no effect on the computed median.

**Returns** Series or DataFrame

Returned type is the same as the original object.

See also:

**Series.expanding** Calling object with Series data

**DataFrame.expanding** Calling object with DataFrames

**Series.median** Equivalent method for Series

**DataFrame.median** Equivalent method for DataFrame

## Examples

Compute the rolling median of a series with a window size of 3.

```
>>> s = pd.Series([0, 1, 2, 3, 4])
>>> s.rolling(3).median()
0      NaN
1      NaN
2      1.0
3      2.0
4      3.0
dtype: float64
```

### 34.16.2.5 pandas.core.window.Expanding.var

Expanding.**var**(*ddof=1, \*args, \*\*kwargs*)

Calculate unbiased expanding variance.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

**Parameters** *ddof*: int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**\*args, \*\*kwargs**

For NumPy compatibility. No additional arguments are used.

**Returns** **Series or DataFrame**

Returns the same object type as the caller of the expanding calculation.

**See also:**

**Series.expanding** Calling object with Series data

**DataFrame.expanding** Calling object with DataFrames

**Series.var** Equivalent method for Series

**DataFrame.var** Equivalent method for DataFrame

**numpy.var** Equivalent method for Numpy array

### Notes

The default *ddof* of 1 used in `Series.var()` is different than the default *ddof* of 0 in `numpy.var()`.

A minimum of 1 period is required for the rolling calculation.

### Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).var()
0      NaN
1      NaN
2    0.333333
3    1.000000
4    1.000000
5    1.333333
```

(continues on next page)

(continued from previous page)

```
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).var()
0      NaN
1      NaN
2    0.333333
3    0.916667
4    0.800000
5    0.700000
6    0.619048
dtype: float64
```

### 34.16.2.6 pandas.core.window.Expanding.std

Expanding.**std**(*ddof=1*, \*args, \*\*kwargs)

Calculate expanding standard deviation.

Normalized by N-1 by default. This can be changed using the *ddof* argument.

**Parameters** *ddof*: int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**\*args, \*\*kwargs**

For NumPy compatibility. No additional arguments are used.

**Returns** **Series or DataFrame**

Returns the same object type as the caller of the expanding calculation.

**See also:**

**Series.expanding** Calling object with Series data

**DataFrame.expanding** Calling object with DataFrames

**Series.std** Equivalent method for Series

**DataFrame.std** Equivalent method for DataFrame

**numpy.std** Equivalent method for Numpy array

#### Notes

The default *ddof* of 1 used in Series.std is different than the default *ddof* of 0 in numpy.std.

A minimum of one period is required for the rolling calculation.

#### Examples

```
>>> s = pd.Series([5, 5, 6, 7, 5, 5, 5])
>>> s.rolling(3).std()
0      NaN
1      NaN
2    0.577350
3    1.000000
4    1.000000
5    1.154701
6    0.000000
dtype: float64
```

```
>>> s.expanding(3).std()
0      NaN
1      NaN
2    0.577350
3    0.957427
4    0.894427
5    0.836660
6    0.786796
dtype: float64
```

### 34.16.2.7 pandas.core.window.Expanding.min

`Expanding.min(*args, **kwargs)`  
Calculate the expanding minimum.

**Parameters** **\*\*kwargs**

Under Review.

**Returns** **Series or DataFrame**

Returned object type is determined by the caller of the expanding calculation.

**See also:**

**Series.expanding** Calling object with a Series

**DataFrame.expanding** Calling object with a DataFrame

**Series.min** Similar method for Series

**DataFrame.min** Similar method for DataFrame

### Examples

Performing a rolling minimum with a window size of 3.

```
>>> s = pd.Series([4, 3, 5, 2, 6])
>>> s.rolling(3).min()
0      NaN
1      NaN
2    3.0
3    2.0
4    2.0
dtype: float64
```



### 34.16.2.8 pandas.core.window.Expanding.max

`Expanding.max(*args, **kwargs)`  
expanding maximum

**Returns**

same type as input

**See also:**

*pandas.Series.expanding, pandas.DataFrame.expanding*

### 34.16.2.9 pandas.core.window.Expanding.corr

`Expanding.corr(other=None, pairwise=None, **kwargs)`  
expanding sample correlation

**Parameters** *other* : Series, DataFrame, or ndarray, optional

if not supplied then will default to self and produce pairwise output

**pairwise** : bool, default None

If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**Returns**

same type as input

**See also:**

*pandas.Series.expanding, pandas.DataFrame.expanding*

### 34.16.2.10 pandas.core.window.Expanding.cov

`Expanding.cov(other=None, pairwise=None, ddof=1, **kwargs)`  
expanding sample covariance

**Parameters** *other* : Series, DataFrame, or ndarray, optional

if not supplied then will default to self and produce pairwise output

**pairwise** : bool, default None

If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndexed DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**ddof** : int, default 1

Delta Degrees of Freedom. The divisor used in calculations is  $N - \text{ddof}$ , where  $N$  represents the number of elements.

**Returns**

same type as input

See also:

*pandas.Series.expanding, pandas.DataFrame.expanding*

#### 34.16.2.11 pandas.core.window.Expanding.skew

`Expanding.skew(**kwargs)`

Unbiased expanding skewness

**Returns**

same type as input

See also:

*pandas.Series.expanding, pandas.DataFrame.expanding*

#### 34.16.2.12 pandas.core.window.Expanding.kurt

`Expanding.kurt(**kwargs)`

Calculate unbiased expanding kurtosis.

This function uses Fisher's definition of kurtosis without bias.

**Parameters** `**kwargs`

Under Review.

**Returns** **Series or DataFrame**

Returned object type is determined by the caller of the expanding calculation

See also:

**Series.expanding** Calling object with Series data

**DataFrame.expanding** Calling object with DataFrames

**Series.kurt** Equivalent method for Series

**DataFrame.kurt** Equivalent method for DataFrame

**scipy.stats.skew** Third moment of a probability density

**scipy.stats.kurtosis** Reference SciPy method

#### Notes

A minimum of 4 periods is required for the expanding calculation.

#### Examples

The example below will show an expanding calculation with a window size of four matching the equivalent function call using *scipy.stats*.

```

>>> arr = [1, 2, 3, 4, 999]
>>> import scipy.stats
>>> fmt = "{0:.6f}" # limit the printed precision to 6 digits
>>> print(fmt.format(scipy.stats.kurtosis(arr[:-1], bias=False)))
-1.200000
>>> print(fmt.format(scipy.stats.kurtosis(arr, bias=False)))
4.999874
>>> s = pd.Series(arr)
>>> s.expanding(4).kurt()
0      NaN
1      NaN
2      NaN
3    -1.200000
4     4.999874
dtype: float64

```

### 34.16.2.13 pandas.core.window.Expanding.apply

`Expanding.apply` (*func*, *raw=None*, *args=()*, *kwargs={}*)

expanding function apply

**Parameters** *func* : function

Must produce a single value from an ndarray input if *raw=True* or a Series if *raw=False*

**raw** : bool, default None

- *False* : passes each row or column as a Series to the function.
- *True* or *None* : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.

The *raw* parameter is required and will show a FutureWarning if not passed. In the future *raw* will default to *False*.

New in version 0.23.0.

**\*args and \*\*kwargs are passed to the function**

**Returns**

same type as input

**See also:**

`pandas.Series.expanding`, `pandas.DataFrame.expanding`

### 34.16.2.14 pandas.core.window.Expanding.aggregate

`Expanding.aggregate` (*arg*, *\*args*, *\*\*kwargs*)

Aggregate using one or more operations over the specified axis.

**Parameters** *func* : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a Series/DataFrame or when passed to Series/DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.

**Returns**

**aggregated** [Series/DataFrame]

**See also:**

`pandas.DataFrame.expanding.aggregate`, `pandas.DataFrame.rolling.aggregate`,  
`pandas.DataFrame.aggregate`

**Notes**

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

**Examples**

```
>>> df = pd.DataFrame(np.random.randn(10, 3), columns=['A', 'B', 'C'])
>>> df
```

	A	B	C
0	-2.385977	-0.102758	0.438822
1	-1.004295	0.905829	-0.954544
2	0.735167	-0.165272	-1.619346
3	-0.702657	-1.340923	-0.706334
4	-0.246845	0.211596	-0.901819
5	2.463718	3.157577	-1.380906
6	-1.142255	2.340594	-0.039875
7	1.396598	-1.647453	1.677227
8	-0.543425	1.761277	-0.220481
9	-0.640505	0.289374	-1.550670

```
>>> df.ewm(alpha=0.5).mean()
```

	A	B	C
0	-2.385977	-0.102758	0.438822
1	-1.464856	0.569633	-0.490089
2	-0.207700	0.149687	-1.135379
3	-0.471677	-0.645305	-0.906555
4	-0.355635	-0.203033	-0.904111
5	1.076417	1.503943	-1.146293
6	-0.041654	1.925562	-0.588728
7	0.680292	0.132049	0.548693

(continues on next page)

(continued from previous page)

```
8  0.067236  0.948257  0.163353
9 -0.286980  0.618493 -0.694496
```

### 34.16.2.15 pandas.core.window.Expanding.quantile

`Expanding.quantile` (*quantile*, *interpolation='linear'*, *\*\*kwargs*)  
expanding quantile.

**Parameters** *quantile* : float

Quantile to compute.  $0 \leq \text{quantile} \leq 1$ .

**interpolation** : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}

New in version 0.23.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points *i* and *j*:

- linear:  $i + (j - i) * \text{fraction}$ , where *fraction* is the fractional part of the index surrounded by *i* and *j*.
- lower: *i*.
- higher: *j*.
- nearest: *i* or *j* whichever is nearest.
- midpoint:  $(i + j) / 2$ .

**\*\*kwargs:**

For compatibility with other expanding methods. Has no effect on the result.

**Returns** **Series or DataFrame**

Returned object type is determined by the caller of the expanding calculation.

**See also:**

[`pandas.Series.quantile`](#) Computes value at the given quantile over all data in Series.

[`pandas.DataFrame.quantile`](#) Computes values at the given quantile over requested axis in DataFrame.

### Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s.rolling(2).quantile(.4, interpolation='lower')
0    NaN
1    1.0
2    2.0
3    3.0
dtype: float64
```

```
>>> s.rolling(2).quantile(.4, interpolation='midpoint')
0    NaN
1    1.5
2    2.5
```

(continues on next page)

(continued from previous page)

```
3      3.5
dtype: float64
```

### 34.16.3 Exponentially-weighted moving window functions

<code>EWM.mean(*args, **kwargs)</code>	exponential weighted moving average
<code>EWM.std([bias])</code>	exponential weighted moving stddev
<code>EWM.var([bias])</code>	exponential weighted moving variance
<code>EWM.corr([other, pairwise])</code>	exponential weighted sample correlation
<code>EWM.cov([other, pairwise, bias])</code>	exponential weighted sample covariance

#### 34.16.3.1 pandas.core.window.EWM.mean

`EWM.mean(*args, **kwargs)`  
exponential weighted moving average

**Returns**

same type as input

See also:

`pandas.Series.ewm`, `pandas.DataFrame.ewm`

#### 34.16.3.2 pandas.core.window.EWM.std

`EWM.std(bias=False, *args, **kwargs)`  
exponential weighted moving stddev

**Parameters** `bias` : boolean, default False

Use a standard estimation bias correction

**Returns**

same type as input

See also:

`pandas.Series.ewm`, `pandas.DataFrame.ewm`

#### 34.16.3.3 pandas.core.window.EWM.var

`EWM.var(bias=False, *args, **kwargs)`  
exponential weighted moving variance

**Parameters** `bias` : boolean, default False

Use a standard estimation bias correction

**Returns**

same type as input

See also:

`pandas.Series.ewm`, `pandas.DataFrame.ewm`

### 34.16.3.4 pandas.core.window.EWM.corr

`EWM.corr` (*other=None, pairwise=None, \*\*kwargs*)  
exponential weighted sample correlation

**Parameters** *other* : Series, DataFrame, or ndarray, optional

if not supplied then will default to self and produce pairwise output

**pairwise** : bool, default None

If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**bias** : boolean, default False

Use a standard estimation bias correction

**Returns**

same type as input

**See also:**

`pandas.Series.ewm`, `pandas.DataFrame.ewm`

### 34.16.3.5 pandas.core.window.EWM.cov

`EWM.cov` (*other=None, pairwise=None, bias=False, \*\*kwargs*)  
exponential weighted sample covariance

**Parameters** *other* : Series, DataFrame, or ndarray, optional

if not supplied then will default to self and produce pairwise output

**pairwise** : bool, default None

If False then only matching columns between self and other will be used and the output will be a DataFrame. If True then all pairwise combinations will be calculated and the output will be a MultiIndex DataFrame in the case of DataFrame inputs. In the case of missing elements, only complete pairwise observations will be used.

**bias** : boolean, default False

Use a standard estimation bias correction

**Returns**

same type as input

**See also:**

`pandas.Series.ewm`, `pandas.DataFrame.ewm`

## 34.17 GroupBy

GroupBy objects are returned by groupby calls: `pandas.DataFrame.groupby()`, `pandas.Series.groupby()`, etc.

### 34.17.1 Indexing, iteration

<code>GroupBy.__iter__()</code>	Groupby iterator
<code>GroupBy.groups</code>	dict {group name -> group labels}
<code>GroupBy.indices</code>	dict {group name -> group indices}
<code>GroupBy.get_group(name[, obj])</code>	Constructs NDFrame from group with provided name

#### 34.17.1.1 `pandas.core.groupby.GroupBy.__iter__`

`GroupBy.__iter__()`

Groupby iterator

##### Returns

Generator yielding sequence of (name, subsetted object)  
for each group

#### 34.17.1.2 `pandas.core.groupby.GroupBy.groups`

`GroupBy.groups`

dict {group name -> group labels}

#### 34.17.1.3 `pandas.core.groupby.GroupBy.indices`

`GroupBy.indices`

dict {group name -> group indices}

#### 34.17.1.4 `pandas.core.groupby.GroupBy.get_group`

`GroupBy.get_group(name, obj=None)`

Constructs NDFrame from group with provided name

**Parameters** `name` : object

the name of the group to get as a DataFrame

`obj` : NDFrame, default None

the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

##### Returns

`group` [type of obj]

<code>Grouper([key, level, freq, axis, sort])</code>	A Grouper allows the user to specify a groupby instruction for a target object
--	--

#### 34.17.1.5 `pandas.Grouper`

**class** `pandas.Grouper` (`key=None, level=None, freq=None, axis=0, sort=False`)

A Grouper allows the user to specify a groupby instruction for a target object



This specification will select a column via the key parameter, or if the level and/or axis parameters are given, a level of the index of the target object.

These are local specifications and will override ‘global’ settings, that is the parameters axis and level which are passed to the groupby itself.

**Parameters** **key** : string, defaults to None

groupby key, which selects the grouping column of the target

**level** : name/number, defaults to None

the level for the target index

**freq** : string / frequency object, defaults to None

This will groupby the specified frequency if the target selection (via key or level) is a datetime-like object. For full specification of available frequencies, please see [here](#).

**axis** [number/name of the axis, defaults to 0]

**sort** : boolean, default to False

whether to sort the resulting labels

**additional kwargs to control time-like groupers (when “freq“ is passed)**

**closed** [closed end of interval; ‘left’ or ‘right’]

**label** [interval boundary to use for labeling; ‘left’ or ‘right’]

**convention** : { ‘start’, ‘end’, ‘e’, ‘s’ }

If grouper is PeriodIndex

**base, loffset**

## Returns

A specification for a groupby instruction

## Examples

Syntactic sugar for `df.groupby('A')`

```
>>> df.groupby(Grouper(key='A'))
```

Specify a resample operation on the column ‘date’

```
>>> df.groupby(Grouper(key='date', freq='60s'))
```

Specify a resample operation on the level ‘date’ on the columns axis with a frequency of 60s

```
>>> df.groupby(Grouper(level='date', freq='60s', axis=1))
```

## Attributes

<b>ax</b>	
<b>groups</b>	

### 34.17.2 Function application

<code>GroupBy.apply(func, *args, **kwargs)</code>	Apply function <code>func</code> group-wise and combine the results together.
<code>GroupBy.aggregate(func, *args, **kwargs)</code>	
<code>GroupBy.transform(func, *args, **kwargs)</code>	
<code>GroupBy.pipe(func, *args, **kwargs)</code>	Apply a function <code>func</code> with arguments to this <code>GroupBy</code> object and return the function's result.

#### 34.17.2.1 pandas.core.groupby.GroupBy.apply

`GroupBy.apply(func, *args, **kwargs)`

Apply function `func` group-wise and combine the results together.

The function passed to `apply` must take a dataframe as its first argument and return a dataframe, a series or a scalar. `apply` will then take care of combining the results back together into a single dataframe or series. `apply` is therefore a highly flexible grouping method.

While `apply` is a very flexible method, its downside is that using it can be quite a bit slower than using more specific methods. Pandas offers a wide range of method that will be much faster than using `apply` for their specific purposes, so try to use them before reaching for `apply`.

**Parameters** `func` : function

A callable that takes a dataframe as its first argument, and returns a dataframe, a series or a scalar. In addition the callable may take positional and keyword arguments

**args, kwargs** : tuple and dict

Optional positional and keyword arguments to pass to `func`

**Returns**

**applied** [Series or DataFrame]

**See also:**

**`pipe`** Apply function to the full `GroupBy` object instead of to each group.

`aggregate`, `transform`

## Notes

In the current implementation `apply` calls `func` twice on the first group to decide whether it can take a fast or slow code path. This can lead to unexpected behavior if `func` has side-effects, as they will take effect twice for the first group.

## Examples

```
>>> df = pd.DataFrame({'A': 'a a b'.split(), 'B': [1,2,3], 'C': [4,6, 5]})
>>> g = df.groupby('A')
```

From `df` above we can see that `g` has two groups, `a`, `b`. Calling `apply` in various ways, we can get different grouping results:

Example 1: below the function passed to `apply` takes a dataframe as its argument and returns a dataframe. `apply` combines the result for each group together into a new dataframe:

```
>>> g.apply(lambda x: x / x.sum())
      B    C
0  0.333333  0.4
1  0.666667  0.6
2  1.000000  1.0
```

Example 2: The function passed to `apply` takes a dataframe as its argument and returns a series. `apply` combines the result for each group together into a new dataframe:

```
>>> g.apply(lambda x: x.max() - x.min())
      B    C
A
a    1    2
b    0    0
```

Example 3: The function passed to `apply` takes a dataframe as its argument and returns a scalar. `apply` combines the result for each group together into a series, including setting the index as appropriate:

```
>>> g.apply(lambda x: x.C.max() - x.B.min())
A
a      5
b      2
dtype: int64
```

### 34.17.2.2 pandas.core.groupby.GroupBy.aggregate

`GroupBy.aggregate(func, *args, **kwargs)`

### 34.17.2.3 pandas.core.groupby.GroupBy.transform

`GroupBy.transform(func, *args, **kwargs)`

### 34.17.2.4 pandas.core.groupby.GroupBy.pipe

`GroupBy.pipe(func, *args, **kwargs)`

Apply a function `func` with arguments to this `GroupBy` object and return the function's result.

New in version 0.21.0.

Use `.pipe` when you want to improve readability by chaining together functions that expect `Series`, `DataFrames`, `GroupBy` or `Resampler` objects. Instead of writing

```
>>> h(g(f(df.groupby('group')), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.groupby('group')
...   .pipe(f)
...   .pipe(g, arg1=a)
...   .pipe(h, arg2=b, arg3=c))
```

which is much more readable.

**Parameters** **func** : callable or tuple of (callable, string)

Function to apply to this GroupBy object or, alternatively, a (callable, data\_keyword) tuple where data\_keyword is a string indicating the keyword of callable that expects the GroupBy object.

**args** : iterable, optional

positional arguments passed into func.

**kwargs** : dict, optional

a dictionary of keyword arguments passed into func.

**Returns**

**object** [the return type of func.]

**See also:**

***pandas.Series.pipe*** Apply a function with arguments to a series

***pandas.DataFrame.pipe*** Apply a function with arguments to a dataframe

***apply*** Apply function to each group instead of to the full GroupBy object.

## Notes

See more [here](#)

## Examples

```
>>> df = pd.DataFrame({'A': 'a b a b'.split(), 'B': [1, 2, 3, 4]})
>>> df
   A  B
0  a  1
1  b  2
2  a  3
3  b  4
```

To get the difference between each groups maximum and minimum value in one pass, you can do

```
>>> df.groupby('A').pipe(lambda x: x.max() - x.min())
   B
A
a  2
b  2
```

### 34.17.3 Computations / Descriptive Stats

<code>GroupBy.all(skipna)</code>	Returns True if all values in the group are truthful, else False
<code>GroupBy.any(skipna)</code>	Returns True if any value in the group is truthful, else False
<code>GroupBy.bfill(limit)</code>	Backward fill the values
<code>GroupBy.count()</code>	Compute count of group, excluding missing values
<code>GroupBy.cumcount(ascending)</code>	Number each item in each group from 0 to the length of that group - 1.
<code>GroupBy.ffill(limit)</code>	Forward fill the values
<code>GroupBy.first(**kwargs)</code>	Compute first of group values
<code>GroupBy.head(n)</code>	Returns first n rows of each group.
<code>GroupBy.last(**kwargs)</code>	Compute last of group values
<code>GroupBy.max(**kwargs)</code>	Compute max of group values
<code>GroupBy.mean(*args, **kwargs)</code>	Compute mean of groups, excluding missing values
<code>GroupBy.median(**kwargs)</code>	Compute median of groups, excluding missing values
<code>GroupBy.min(**kwargs)</code>	Compute min of group values
<code>GroupBy.ngroup(ascending)</code>	Number each group from 0 to the number of groups - 1.
<code>GroupBy.nth(n[, dropna])</code>	Take the nth row from each group if n is an int, or a subset of rows if n is a list of ints.
<code>GroupBy.ohlc()</code>	Compute sum of values, excluding missing values For multiple groupings, the result index will be a MultiIndex
<code>GroupBy.prod(**kwargs)</code>	Compute prod of group values
<code>GroupBy.rank(method, ascending, na_option, ...)</code>	Provides the rank of values within each group.
<code>GroupBy.pct_change(periods, fill_method, ...)</code>	Calculate pct_change of each value to previous entry in group
<code>GroupBy.size()</code>	Compute group sizes
<code>GroupBy.sem(ddof)</code>	Compute standard error of the mean of groups, excluding missing values
<code>GroupBy.std(ddof)</code>	Compute standard deviation of groups, excluding missing values
<code>GroupBy.sum(**kwargs)</code>	Compute sum of group values
<code>GroupBy.var(ddof)</code>	Compute variance of groups, excluding missing values
<code>GroupBy.tail(n)</code>	Returns last n rows of each group

#### 34.17.3.1 pandas.core.groupby.GroupBy.all

`GroupBy.all(skipna=True)`

Returns True if all values in the group are truthful, else False

**Parameters** `skipna` : bool, default True

Flag to ignore nan values during truth testing

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

#### 34.17.3.2 pandas.core.groupby.GroupBy.any

`GroupBy.any(skipna=True)`

Returns True if any value in the group is truthful, else False

**Parameters** `skipna` : bool, default True

Flag to ignore nan values during truth testing

**See also:**

*`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`*

### 34.17.3.3 `pandas.core.groupby.GroupBy.bfill`

`GroupBy.bfill` (*limit=None*)

Backward fill the values

**Parameters** `limit` : integer, optional

limit of how many values to fill

**See also:**

`Series.backfill`, `DataFrame.backfill`, `Series.fillna`, `DataFrame.fillna`

### 34.17.3.4 `pandas.core.groupby.GroupBy.count`

`GroupBy.count` ()

Compute count of group, excluding missing values

**See also:**

*`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`*

### 34.17.3.5 `pandas.core.groupby.GroupBy.cumcount`

`GroupBy.cumcount` (*ascending=True*)

Number each item in each group from 0 to the length of that group - 1.

Essentially this is equivalent to

```
>>> self.apply(lambda x: Series(np.arange(len(x)), x.index))
```

**Parameters** `ascending` : bool, default True

If False, number in reverse, from length of group - 1 to 0.

**See also:**

*`ngroup`* Number the groups themselves.

### Examples

```
>>> df = pd.DataFrame([[ 'a'], [ 'a'], [ 'a'], [ 'b'], [ 'b'], [ 'a']],
...                    columns=[ 'A'])
>>> df
   A
0  a
1  a
2  a
```

(continues on next page)

(continued from previous page)

```

3  b
4  b
5  a
>>> df.groupby('A').cumcount()
0    0
1    1
2    2
3    0
4    1
5    3
dtype: int64
>>> df.groupby('A').cumcount(ascending=False)
0    3
1    2
2    1
3    1
4    0
5    0
dtype: int64

```

### 34.17.3.6 pandas.core.groupby.GroupBy.fffll

GroupBy.fffll (*limit=None*)

Forward fill the values

**Parameters** **limit** : integer, optional

limit of how many values to fill

**See also:**

`Series.pad`, `DataFrame.pad`, `Series.fillna`, `DataFrame.fillna`

### 34.17.3.7 pandas.core.groupby.GroupBy.first

GroupBy.first (\*\*kwargs)

Compute first of group values

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

### 34.17.3.8 pandas.core.groupby.GroupBy.head

GroupBy.head (*n=5*)

Returns first n rows of each group.

Essentially equivalent to `.apply(lambda x: x.head(n))`, except ignores `as_index` flag.

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

## Examples

```
>>> df = DataFrame([[1, 2], [1, 4], [5, 6]],
                    columns=['A', 'B'])
>>> df.groupby('A', as_index=False).head(1)
   A  B
0  1  2
2  5  6
>>> df.groupby('A').head(1)
   A  B
0  1  2
2  5  6
```

### 34.17.3.9 pandas.core.groupby.GroupBy.last

GroupBy.**last** (\*\*kwargs)  
Compute last of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.10 pandas.core.groupby.GroupBy.max

GroupBy.**max** (\*\*kwargs)  
Compute max of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.11 pandas.core.groupby.GroupBy.mean

GroupBy.**mean** (\*args, \*\*kwargs)  
Compute mean of groups, excluding missing values  
For multiple groupings, the result index will be a MultiIndex

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.12 pandas.core.groupby.GroupBy.median

GroupBy.**median** (\*\*kwargs)  
Compute median of groups, excluding missing values  
For multiple groupings, the result index will be a MultiIndex

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*



### 34.17.3.13 pandas.core.groupby.GroupBy.min

GroupBy.min(\*\*kwargs)

Compute min of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.14 pandas.core.groupby.GroupBy.ngroup

GroupBy.ngroup(ascending=True)

Number each group from 0 to the number of groups - 1.

This is the enumerative complement of cumcount. Note that the numbers given to the groups match the order in which the groups would be seen when iterating over the groupby object, not the order they are first observed.

New in version 0.20.2.

**Parameters** *ascending* : bool, default True

If False, number in reverse, from number of group - 1 to 0.

**See also:**

*cumcount* Number the rows in each group.

#### Examples

```

>>> df = pd.DataFrame({"A": list("aaabba")})
>>> df
   A
0  a
1  a
2  a
3  b
4  b
5  a
>>> df.groupby('A').ngroup()
0    0
1    0
2    0
3    1
4    1
5    0
dtype: int64
>>> df.groupby('A').ngroup(ascending=False)
0    1
1    1
2    1
3    0
4    0
5    1
dtype: int64
>>> df.groupby(["A", [1,1,2,3,2,1]]).ngroup()
0    0
1    0

```

(continues on next page)

(continued from previous page)

```

2    1
3    3
4    2
5    0
dtype: int64

```

### 34.17.3.15 pandas.core.groupby.GroupBy.nth

`GroupBy.nth(n, dropna=None)`

Take the *nth* row from each group if *n* is an int, or a subset of rows if *n* is a list of ints.

If *dropna*, will take the *nth* non-null row, *dropna* is either Truthy (if a Series) or 'all', 'any' (if a DataFrame); this is equivalent to calling `dropna(how=dropna)` before the groupby.

**Parameters** *n* : int or list of ints

a single *nth* value for the row or a list of *nth* values

**dropna** : None or str, optional

apply the specified *dropna* operation before counting which row is the *nth* row. Needs to be None, 'any' or 'all'

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

### Examples

```

>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                     'B': [np.nan, 2, 3, 4, 5]}, columns=['A', 'B'])
>>> g = df.groupby('A')
>>> g.nth(0)
      B
A
1  NaN
2  3.0
>>> g.nth(1)
      B
A
1  2.0
2  5.0
>>> g.nth(-1)
      B
A
1  4.0
2  5.0
>>> g.nth([0, 1])
      B
A
1  NaN
1  2.0
2  3.0
2  5.0

```

Specifying *dropna* allows count ignoring NaN

```
>>> g.nth(0, dropna='any')
      B
A
1  2.0
2  3.0
```

NaNs denote group exhausted when using dropna

```
>>> g.nth(3, dropna='any')
      B
A
1  NaN
2  NaN
```

Specifying `as_index=False` in `groupby` keeps the original index.

```
>>> df.groupby('A', as_index=False).nth(1)
      A      B
1  1  2.0
4  2  5.0
```

### 34.17.3.16 pandas.core.groupby.GroupBy.ohlc

`GroupBy.ohlc()`

Compute sum of values, excluding missing values For multiple groupings, the result index will be a `MultiIndex`

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.17 pandas.core.groupby.GroupBy.prod

`GroupBy.prod(**kwargs)`

Compute prod of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.18 pandas.core.groupby.GroupBy.rank

`GroupBy.rank(method='average', ascending=True, na_option='keep', pct=False, axis=0)`

Provides the rank of values within each group.

**Parameters** `method`: {'average', 'min', 'max', 'first', 'dense'}, default 'average'

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**ascending**: boolean, default True

False for ranks by high (1) to low (N)

**na\_option** : { 'keep', 'top', 'bottom' }, default 'keep'

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

**pct** : boolean, default False

Compute percentage rank of data within each group

**axis** : int, default 0

The axis of the object over which to compute the rank.

**Returns**

—

**DataFrame with ranking of values within each group**

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.17.3.19 pandas.core.groupby.GroupBy.pct\_change

`GroupBy.pct_change` (*periods=1, fill\_method='pad', limit=None, freq=None, axis=0*)

Calculate pct\_change of each value to previous entry in group

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.17.3.20 pandas.core.groupby.GroupBy.size

`GroupBy.size` ()

Compute group sizes

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.17.3.21 pandas.core.groupby.GroupBy.sem

`GroupBy.sem` (*ddof=1*)

Compute standard error of the mean of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

**Parameters** **ddof** : integer, default 1

degrees of freedom

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.22 pandas.core.groupby.GroupBy.std

GroupBy.**std**(*ddof=1, \*args, \*\*kwargs*)

Compute standard deviation of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

**Parameters** **ddof**: integer, default 1  
degrees of freedom

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.23 pandas.core.groupby.GroupBy.sum

GroupBy.**sum**(*\*\*kwargs*)

Compute sum of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.24 pandas.core.groupby.GroupBy.var

GroupBy.**var**(*ddof=1, \*args, \*\*kwargs*)

Compute variance of groups, excluding missing values

For multiple groupings, the result index will be a MultiIndex

**Parameters** **ddof**: integer, default 1  
degrees of freedom

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.25 pandas.core.groupby.GroupBy.tail

GroupBy.**tail**(*n=5*)

Returns last n rows of each group

Essentially equivalent to `.apply(lambda x: x.tail(n))`, except ignores `as_index` flag.

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### Examples

```

>>> df = DataFrame([['a', 1], ['a', 2], ['b', 1], ['b', 2]],
                    columns=['A', 'B'])
>>> df.groupby('A').tail(1)
   A  B
1  a  2
3  b  2

```

(continues on next page)

(continued from previous page)

```
>>> df.groupby('A').head(1)
   A  B
0  a  1
2  b  1
```

The following methods are available in both `SeriesGroupBy` and `DataFrameGroupBy` objects, but may differ slightly, usually in that the `DataFrameGroupBy` version usually permits the specification of an axis argument, and often an argument indicating whether to restrict application to columns of a specific data type.

<code>DataFrameGroupBy.agg(arg, *args, **kwargs)</code>	Aggregate using one or more operations over the specified axis.
<code>DataFrameGroupBy.all([skipna])</code>	Returns True if all values in the group are truthful, else False
<code>DataFrameGroupBy.any([skipna])</code>	Returns True if any value in the group is truthful, else False
<code>DataFrameGroupBy.bfill([limit])</code>	Backward fill the values
<code>DataFrameGroupBy.corr</code>	Compute pairwise correlation of columns, excluding NA/null values
<code>DataFrameGroupBy.count()</code>	Compute count of group, excluding missing values
<code>DataFrameGroupBy.cov</code>	Compute pairwise covariance of columns, excluding NA/null values.
<code>DataFrameGroupBy.cummax([axis])</code>	Cumulative max for each group
<code>DataFrameGroupBy.cummin([axis])</code>	Cumulative min for each group
<code>DataFrameGroupBy.cumprod([axis])</code>	Cumulative product for each group
<code>DataFrameGroupBy.cumsum([axis])</code>	Cumulative sum for each group
<code>DataFrameGroupBy.describe(**kwargs)</code>	Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.
<code>DataFrameGroupBy.diff</code>	First discrete difference of element.
<code>DataFrameGroupBy.ffill([limit])</code>	Forward fill the values
<code>DataFrameGroupBy.fillna</code>	Fill NA/NaN values using the specified method
<code>DataFrameGroupBy.filter(func[, dropna])</code>	Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by func.
<code>DataFrameGroupBy.hist</code>	Make a histogram of the DataFrame's.
<code>DataFrameGroupBy.idxmax</code>	Return index of first occurrence of maximum over requested axis.
<code>DataFrameGroupBy.idxmin</code>	Return index of first occurrence of minimum over requested axis.
<code>DataFrameGroupBy.mad</code>	Return the mean absolute deviation of the values for the requested axis
<code>DataFrameGroupBy.pct_change([periods, ...])</code>	Calculate pct_change of each value to previous entry in group
<code>DataFrameGroupBy.plot</code>	Class implementing the .plot attribute for groupby objects
<code>DataFrameGroupBy.quantile</code>	Return values at the given quantile over requested axis, a la numpy.percentile.
<code>DataFrameGroupBy.rank([method, ascending, ...])</code>	Provides the rank of values within each group.
<code>DataFrameGroupBy.resample(rule, *args, **kwargs)</code>	Provide resampling when using a TimeGrouper Return a new grouper with our resampler appended

Continued on next page

Table 167 – continued from previous page

<code>DataFrameGroupBy.shift([periods, freq, axis])</code>	Shift each group by periods observations
<code>DataFrameGroupBy.size()</code>	Compute group sizes
<code>DataFrameGroupBy.skew</code>	Return unbiased skew over requested axis Normalized by N-1
<code>DataFrameGroupBy.take</code>	Return the elements in the given <i>positional</i> indices along an axis.
<code>DataFrameGroupBy.tshift</code>	Shift the time index, using the index's frequency if available.

### 34.17.3.26 pandas.core.groupby.DataFrameGroupBy.agg

`DataFrameGroupBy.agg` (*arg*, \**args*, \*\**kwargs*)

Aggregate using one or more operations over the specified axis.

**Parameters** *func* : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a `DataFrame` or when passed to `DataFrame.apply`. For a `DataFrame`, can pass a dict, if the keys are `DataFrame` column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.

**Returns**

**aggregated** [`DataFrame`]

**See also:**

`pandas.DataFrame.groupby.apply`, `pandas.DataFrame.groupby.transform`, [`pandas.DataFrame.aggregate`](#)

#### Notes

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a `Series` for evaluation.

#### Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 2],
...                     'B': [1, 2, 3, 4],
...                     'C': np.random.randn(4)})
```

```
>>> df
   A  B      C
0  1  1  0.362838
1  1  2  0.227877
2  2  3  1.267767
3  2  4 -0.562860
```

The aggregation is for each column.

```
>>> df.groupby('A').agg('min')
   B      C
A
1  1  0.227877
2  3 -0.562860
```

### Multiple aggregations

```
>>> df.groupby('A').agg(['min', 'max'])
   B      C
   min max   min   max
A
1  1  2  0.227877  0.362838
2  3  4 -0.562860  1.267767
```

### Select a column for aggregation

```
>>> df.groupby('A').B.agg(['min', 'max'])
   min  max
A
1     1     2
2     3     4
```

### Different aggregations per column

```
>>> df.groupby('A').agg({'B': ['min', 'max'], 'C': 'sum'})
   B      C
   min max   sum
A
1  1  2  0.590716
2  3  4  0.704907
```

### 34.17.3.27 pandas.core.groupby.DataFrameGroupBy.all

`DataFrameGroupBy.all` (*skipna=True*)

Returns True if all values in the group are truthful, else False

**Parameters** `skipna` : bool, default True

Flag to ignore nan values during truth testing

**See also:**

[`pandas.Series.groupby`](#), [`pandas.DataFrame.groupby`](#), [`pandas.Panel.groupby`](#)



**34.17.3.28 pandas.core.groupby.DataFrameGroupBy.any**

DataFrameGroupBy.**any** (*skipna=True*)

Returns True if any value in the group is truthful, else False

**Parameters** **skipna** : bool, default True

Flag to ignore nan values during truth testing

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

**34.17.3.29 pandas.core.groupby.DataFrameGroupBy.bfill**

DataFrameGroupBy.**bfill** (*limit=None*)

Backward fill the values

**Parameters** **limit** : integer, optional

limit of how many values to fill

**See also:**

*Series.backfill, DataFrame.backfill, Series.fillna, DataFrame.fillna*

**34.17.3.30 pandas.core.groupby.DataFrameGroupBy.corr**

DataFrameGroupBy.**corr**

Compute pairwise correlation of columns, excluding NA/null values

**Parameters** **method** : {'pearson', 'kendall', 'spearman'}

- pearson : standard correlation coefficient
- kendall : Kendall Tau correlation coefficient
- spearman : Spearman rank correlation

**min\_periods** : int, optional

Minimum number of observations required per pair of columns to have a valid result.  
Currently only available for pearson and spearman correlation

**Returns**

y [DataFrame]

**34.17.3.31 pandas.core.groupby.DataFrameGroupBy.count**

DataFrameGroupBy.**count** ()

Compute count of group, excluding missing values

**34.17.3.32 pandas.core.groupby.DataFrameGroupBy.cov**

DataFrameGroupBy.**cov**

Compute pairwise covariance of columns, excluding NA/null values.

Compute the pairwise covariance among the series of a DataFrame. The returned data frame is the **covariance matrix** of the columns of the DataFrame.

Both NA and null values are automatically excluded from the calculation. (See the note below about bias from missing values.) A threshold can be set for the minimum number of observations for each value created. Comparisons with observations below this threshold will be returned as NaN.

This method is generally used for the analysis of time series data to understand the relationship between different measures across time.

**Parameters** `min_periods` : int, optional

Minimum number of observations required per pair of columns to have a valid result.

**Returns** **DataFrame**

The covariance matrix of the series of the DataFrame.

**See also:**

`pandas.Series.cov` compute covariance with another Series

`pandas.core.window.EWM.cov` exponential weighted sample covariance

`pandas.core.window.Expanding.cov` expanding sample covariance

`pandas.core.window.Rolling.cov` rolling sample covariance

## Notes

Returns the covariance matrix of the DataFrame's time series. The covariance is normalized by N-1.

For DataFrames that have Series that are missing data (assuming that data is [missing at random](#)) the returned covariance matrix will be an unbiased estimate of the variance and covariance between the member Series.

However, for many applications this estimate may not be acceptable because the estimate covariance matrix is not guaranteed to be positive semi-definite. This could lead to estimate correlations having absolute values which are greater than one, and/or a non-invertible covariance matrix. See [Estimation of covariance matrices](#) for more details.

## Examples

```
>>> df = pd.DataFrame([(1, 2), (0, 3), (2, 0), (1, 1)],
...                    columns=['dogs', 'cats'])
>>> df.cov()
           dogs      cats
dogs  0.666667 -1.000000
cats -1.000000  1.666667
```

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(1000, 5),
...                    columns=['a', 'b', 'c', 'd', 'e'])
>>> df.cov()
           a          b          c          d          e
a  0.998438 -0.020161  0.059277 -0.008943  0.014144
b -0.020161  1.059352 -0.008543 -0.024738  0.009826
c  0.059277 -0.008543  1.010670 -0.001486 -0.000271
d -0.008943 -0.024738 -0.001486  0.921297 -0.013692
e  0.014144  0.009826 -0.000271 -0.013692  0.977795
```

**Minimum number of periods**

This method also supports an optional `min_periods` keyword that specifies the required minimum number of non-NA observations for each column pair in order to have a valid result:

```
>>> np.random.seed(42)
>>> df = pd.DataFrame(np.random.randn(20, 3),
...                    columns=['a', 'b', 'c'])
>>> df.loc[df.index[:5], 'a'] = np.nan
>>> df.loc[df.index[5:10], 'b'] = np.nan
>>> df.cov(min_periods=12)
           a          b          c
a  0.316741      NaN -0.150812
b      NaN  1.248003  0.191417
c -0.150812  0.191417  0.895202
```

**34.17.3.33 pandas.core.groupby.DataFrameGroupBy.cummax**

`DataFrameGroupBy.cummax` (*axis=0, \*\*kwargs*)

Cumulative max for each group

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

**34.17.3.34 pandas.core.groupby.DataFrameGroupBy.cummin**

`DataFrameGroupBy.cummin` (*axis=0, \*\*kwargs*)

Cumulative min for each group

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

**34.17.3.35 pandas.core.groupby.DataFrameGroupBy.cumprod**

`DataFrameGroupBy.cumprod` (*axis=0, \*args, \*\*kwargs*)

Cumulative product for each group

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

**34.17.3.36 pandas.core.groupby.DataFrameGroupBy.cumsum**

`DataFrameGroupBy.cumsum` (*axis=0, \*args, \*\*kwargs*)

Cumulative sum for each group

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.37 pandas.core.groupby.DataFrameGroupBy.describe

`DataFrameGroupBy.describe` (\*\*kwargs)

Generates descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN values.

Analyzes both numeric and object series, as well as `DataFrame` column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

**Parameters** `percentiles` : list-like of numbers, optional

The percentiles to include in the output. All should fall between 0 and 1. The default is `[.25, .5, .75]`, which returns the 25th, 50th, and 75th percentiles.

**include** : 'all', list-like of dtypes or None (default), optional

A white list of data types to include in the result. Ignored for `Series`. Here are the options:

- 'all' : All columns of the input will be included in the output.
- A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit `numpy.number`. To limit it instead to object columns submit the `numpy.object` data type. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To select pandas categorical columns, use 'category'
- None (default) : The result will include all numeric columns.

**exclude** : list-like of dtypes or None (default), optional,

A black list of data types to omit from the result. Ignored for `Series`. Here are the options:

- A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit `numpy.number`. To exclude object columns submit the data type `numpy.object`. Strings can also be used in the style of `select_dtypes` (e.g. `df.describe(include=['O'])`). To exclude pandas categorical columns, use 'category'
- None (default) : The result will exclude nothing.

**Returns**

**summary:** `Series/DataFrame` of summary statistics

**See also:**

`DataFrame.count`, `DataFrame.max`, `DataFrame.min`, `DataFrame.mean`, `DataFrame.std`, `DataFrame.select_dtypes`

#### Notes

For numeric data, the result's index will include `count`, `mean`, `std`, `min`, `max` as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result's index will include `count`, `unique`, `top`, and `freq`. The `top` is the most common value. The `freq` is the most common value's frequency. Timestamps also include the `first` and `last` items.

If multiple object values have the highest count, then the `count` and `top` results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a `DataFrame`, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns. If `include='all'` is provided as an option, the result will include a union of attributes of each type.

The `include` and `exclude` parameters can be used to limit which columns in a `DataFrame` are analyzed for the output. The parameters are ignored when analyzing a `Series`.

## Examples

Describing a numeric Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count      4
unique     3
top        a
freq       2
dtype: object
```

Describing a timestamp Series.

```
>>> s = pd.Series([
...     np.datetime64("2000-01-01"),
...     np.datetime64("2010-01-01"),
...     np.datetime64("2010-01-01")
... ])
>>> s.describe()
count      3
unique     2
top        2010-01-01 00:00:00
freq       2
first      2000-01-01 00:00:00
last       2010-01-01 00:00:00
dtype: object
```

Describing a `DataFrame`. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'object': ['a', 'b', 'c'],
...                    'numeric': [1, 2, 3],
...                    'categorical': pd.Categorical(['d', 'e', 'f'])
...                    })
```

(continues on next page)

(continued from previous page)

```
...    })
>>> df.describe()
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing all columns of a DataFrame regardless of data type.

```
>>> df.describe(include='all')
      categorical  numeric  object
count           3      3.0      3
unique          3      NaN      3
top             f      NaN      c
freq            1      NaN      1
mean           NaN      2.0     NaN
std            NaN      1.0     NaN
min            NaN      1.0     NaN
25%            NaN      1.5     NaN
50%            NaN      2.0     NaN
75%            NaN      2.5     NaN
max            NaN      3.0     NaN
```

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
Name: numeric, dtype: float64
```

Including only numeric columns in a DataFrame description.

```
>>> df.describe(include=[np.number])
      numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Including only string columns in a DataFrame description.

```
>>> df.describe(include=[np.object])
      object
count      3
unique     3
top        c
freq       1
```

Including only categorical columns from a DataFrame description.

```
>>> df.describe(include=['category'])
      categorical
count          3
unique         3
top            f
freq           1
```

Excluding numeric columns from a DataFrame description.

```
>>> df.describe(exclude=[np.number])
      categorical  object
count           3      3
unique          3      3
top            f      c
freq           1      1
```

Excluding object columns from a DataFrame description.

```
>>> df.describe(exclude=[np.object])
      categorical  numeric
count           3      3.0
unique          3      NaN
top            f      NaN
freq           1      NaN
mean           NaN      2.0
std            NaN      1.0
min            NaN      1.0
25%            NaN      1.5
50%            NaN      2.0
75%            NaN      2.5
max            NaN      3.0
```

### 34.17.3.38 pandas.core.groupby.DataFrameGroupBy.diff

DataFrameGroupBy.**diff**

First discrete difference of element.

Calculates the difference of a DataFrame element compared with another element in the DataFrame (default is the element in the same column of the previous row).

**Parameters** **periods** : int, default 1

Periods to shift for calculating difference, accepts negative values.

**axis** : {0 or 'index', 1 or 'columns'}, default 0

Take difference over rows (0) or columns (1).

New in version 0.16.1..

**Returns****diffed** [DataFrame]**See also:****Series.diff** First discrete difference for a Series.**DataFrame.pct\_change** Percent change over given number of periods.**DataFrame.shift** Shift index by desired number of periods with an optional time freq.**Examples**

Difference with previous row

```
>>> df = pd.DataFrame({'a': [1, 2, 3, 4, 5, 6],
...                    'b': [1, 1, 2, 3, 5, 8],
...                    'c': [1, 4, 9, 16, 25, 36]})
>>> df
   a  b  c
0  1  1  1
1  2  1  4
2  3  2  9
3  4  3 16
4  5  5 25
5  6  8 36
```

```
>>> df.diff()
   a    b    c
0 NaN NaN NaN
1 1.0 0.0 3.0
2 1.0 1.0 5.0
3 1.0 1.0 7.0
4 1.0 2.0 9.0
5 1.0 3.0 11.0
```

Difference with previous column

```
>>> df.diff(axis=1)
   a    b    c
0 NaN 0.0 0.0
1 NaN -1.0 3.0
2 NaN -1.0 7.0
3 NaN -1.0 13.0
4 NaN 0.0 20.0
5 NaN 2.0 28.0
```

Difference with 3rd previous row

```
>>> df.diff(periods=3)
   a    b    c
0 NaN NaN NaN
1 NaN NaN NaN
2 NaN NaN NaN
3 3.0 2.0 15.0
4 3.0 4.0 21.0
5 3.0 6.0 27.0
```



Difference with following row

```
>>> df.diff(periods=-1)
      a      b      c
0 -1.0   0.0  -3.0
1 -1.0  -1.0  -5.0
2 -1.0  -1.0  -7.0
3 -1.0  -2.0  -9.0
4 -1.0  -3.0 -11.0
5   NaN   NaN   NaN
```

### 34.17.3.39 pandas.core.groupby.DataFrameGroupBy.fffll

DataFrameGroupBy.fffll (*limit=None*)

Forward fill the values

**Parameters** *limit* : integer, optional

limit of how many values to fill

**See also:**

Series.pad, DataFrame.pad, Series.fillna, DataFrame.fillna

### 34.17.3.40 pandas.core.groupby.DataFrameGroupBy.fillna

DataFrameGroupBy.fillna

Fill NA/NaN values using the specified method

**Parameters** *value* : scalar, dict, Series, or DataFrame

Value to use to fill holes (e.g. 0), alternately a dict/Series/DataFrame of values specifying which value to use for each index (for a Series) or column (for a DataFrame). (values not in the dict/Series/DataFrame will not be filled). This value cannot be a list.

**method** : {'backfill', 'bfill', 'pad', 'ffill', None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**axis** [{0 or 'index', 1 or 'columns'}]

**inplace** : boolean, default False

If True, fill in place. Note: this will modify any other views on this object, (e.g. a no-copy slice for a column in a DataFrame).

**limit** : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled. Must be greater than 0 if not None.

**downcast** : dict, default is None

a dict of item->dtype of what to downcast if possible, or the string 'infer' which will try to downcast to an appropriate equal type (e.g. float64 to int64 if possible)

**Returns**

**filled** [DataFrame]

**See also:**

**interpolate** Fill NaN values using interpolation.

reindex, asfreq

**Examples**

```
>>> df = pd.DataFrame([[np.nan, 2, np.nan, 0],
...                    [3, 4, np.nan, 1],
...                    [np.nan, np.nan, np.nan, 5],
...                    [np.nan, 3, np.nan, 4]],
...                    columns=list('ABCD'))
>>> df
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2 NaN  NaN NaN  5
3 NaN  3.0 NaN  4
```

Replace all NaN elements with 0s.

```
>>> df.fillna(0)
   A    B    C    D
0  0.0  2.0  0.0  0
1  3.0  4.0  0.0  1
2  0.0  0.0  0.0  5
3  0.0  3.0  0.0  4
```

We can also propagate non-null values forward or backward.

```
>>> df.fillna(method='ffill')
   A    B    C    D
0 NaN  2.0 NaN  0
1  3.0  4.0 NaN  1
2  3.0  4.0 NaN  5
3  3.0  3.0 NaN  4
```

Replace all NaN elements in column 'A', 'B', 'C', and 'D', with 0, 1, 2, and 3 respectively.

```
>>> values = {'A': 0, 'B': 1, 'C': 2, 'D': 3}
>>> df.fillna(value=values)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  2.0  1
2  0.0  1.0  2.0  5
3  0.0  3.0  2.0  4
```

Only replace the first NaN element.

```
>>> df.fillna(value=values, limit=1)
   A    B    C    D
0  0.0  2.0  2.0  0
1  3.0  4.0  NaN  1
2  NaN  1.0  NaN  5
3  NaN  3.0  NaN  4
```

### 34.17.3.41 pandas.core.groupby.DataFrameGroupBy.filter

DataFrameGroupBy.**filter** (*func*, *dropna=True*, *\*args*, *\*\*kwargs*)

Return a copy of a DataFrame excluding elements from groups that do not satisfy the boolean criterion specified by *func*.

**Parameters** *f*: function

Function to apply to each subframe. Should return True or False.

**dropna**: Drop groups that do not pass the filter. True by default;  
if False, groups that evaluate False are filled with NaNs.

**Returns**

**filtered** [DataFrame]

#### Notes

Each subframe is endowed the attribute ‘name’ in case you need to know which group you are working on.

#### Examples

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
...                          'foo', 'bar'],
...                   'B' : [1, 2, 3, 4, 5, 6],
...                   'C' : [2.0, 5., 8., 1., 2., 9.]})
>>> grouped = df.groupby('A')
>>> grouped.filter(lambda x: x['B'].mean() > 3.)
   A    B    C
1  bar  2  5.0
3  bar  4  1.0
5  bar  6  9.0
```

### 34.17.3.42 pandas.core.groupby.DataFrameGroupBy.hist

DataFrameGroupBy.**hist**

Make a histogram of the DataFrame’s.

A **histogram** is a representation of the distribution of data. This function calls `matplotlib.pyplot.hist()`, on each series in the DataFrame, resulting in one histogram per column.

**Parameters** *data*: DataFrame

The pandas object holding the data.

**column** : string or sequence

If passed, will be used to limit data to a subset of columns.

**by** : object, optional

If passed, then used to form histograms for separate groups.

**grid** : boolean, default True

Whether to show axis grid lines.

**xlabelsize** : int, default None

If specified changes the x-axis label size.

**xrot** : float, default None

Rotation of x axis labels. For example, a value of 90 displays the x labels rotated 90 degrees clockwise.

**ylabelsize** : int, default None

If specified changes the y-axis label size.

**yrot** : float, default None

Rotation of y axis labels. For example, a value of 90 displays the y labels rotated 90 degrees clockwise.

**ax** : Matplotlib axes object, default None

The axes to plot the histogram on.

**sharex** : boolean, default True if ax is None else False

In case subplots=True, share x axis and set some x axis labels to invisible; defaults to True if ax is None otherwise False if an ax is passed in. Note that passing in both an ax and sharex=True will alter all x axis labels for all subplots in a figure.

**sharey** : boolean, default False

In case subplots=True, share y axis and set some y axis labels to invisible.

**figsize** : tuple

The size in inches of the figure to create. Uses the value in *matplotlib.rcParams* by default.

**layout** : tuple, optional

Tuple of (rows, columns) for the layout of the histograms.

**bins** : integer or sequence, default 10

Number of histogram bins to be used. If an integer is given, bins + 1 bin edges are calculated and returned. If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

**\*\*kwds**

All other plotting keyword arguments to be passed to `matplotlib.pyplot.hist()`.

#### Returns

**axes** [matplotlib.AxesSubplot or numpy.ndarray of them]

See also:

`matplotlib.pyplot.hist` Plot a histogram using matplotlib.

## Examples

This example draws a histogram based on the length and width of some animals, displayed in three bins

```
>>> df = pd.DataFrame({
...     'length': [1.5, 0.5, 1.2, 0.9, 3],
...     'width': [0.7, 0.2, 0.15, 0.2, 1.1]
... }, index= ['pig', 'rabbit', 'duck', 'chicken', 'horse'])
>>> hist = df.hist(bins=3)
```

### 34.17.3.43 pandas.core.groupby.DataFrameGroupBy.idxmax

DataFrameGroupBy.**idxmax**

Return index of first occurrence of maximum over requested axis. NA/null values are excluded.

**Parameters** `axis` : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

**idxmax** [Series]

**Raises** **ValueError**

- If the row/column is empty

**See also:**

`Series.idxmax`

## Notes

This method is the DataFrame version of `ndarray.argmax`.

### 34.17.3.44 pandas.core.groupby.DataFrameGroupBy.idxmin

DataFrameGroupBy.**idxmin**

Return index of first occurrence of minimum over requested axis. NA/null values are excluded.

**Parameters** `axis` : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA.

**Returns**

**idxmin** [Series]

**Raises** **ValueError**

- If the row/column is empty

**See also:**

`Series.idxmin`

## Notes

This method is the DataFrame version of `ndarray.argmax`.

### 34.17.3.45 `pandas.core.groupby.DataFrameGroupBy.mad`

`DataFrameGroupBy.mad`

Return the mean absolute deviation of the values for the requested axis

#### Parameters

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

#### Returns

**mad** [Series or DataFrame (if level specified)]

### 34.17.3.46 `pandas.core.groupby.DataFrameGroupBy.pct_change`

`DataFrameGroupBy.pct_change` (*periods=1, fill\_method='pad', limit=None, freq=None, axis=0*)

Calculate pct\_change of each value to previous entry in group

**See also:**

`pandas.Series.groupby`, `pandas.DataFrame.groupby`, `pandas.Panel.groupby`

### 34.17.3.47 `pandas.core.groupby.DataFrameGroupBy.plot`

`DataFrameGroupBy.plot`

Class implementing the .plot attribute for groupby objects

### 34.17.3.48 `pandas.core.groupby.DataFrameGroupBy.quantile`

`DataFrameGroupBy.quantile`

Return values at the given quantile over requested axis, a la numpy.percentile.

**Parameters** **q** : float or array-like, default 0.5 (50% quantile)

$0 \leq q \leq 1$ , the quantile(s) to compute

**axis** : {0, 1, 'index', 'columns'} (default 0)

0 or 'index' for row-wise, 1 or 'columns' for column-wise

**numeric\_only** : boolean, default True

If False, the quantile of datetime and timedelta data will be computed as well

**interpolation** : {'linear', 'lower', 'higher', 'midpoint', 'nearest'}

New in version 0.18.0.

This optional parameter specifies the interpolation method to use, when the desired quantile lies between two data points  $i$  and  $j$ :

- linear:  $i + (j - i) * \text{fraction}$ , where *fraction* is the fractional part of the index surrounded by  $i$  and  $j$ .
- lower:  $i$ .
- higher:  $j$ .
- nearest:  $i$  or  $j$  whichever is nearest.
- midpoint:  $(i + j) / 2$ .

**Returns quantiles** : Series or DataFrame

- If  $q$  is an array, a DataFrame will be returned where the index is  $q$ , the columns are the columns of self, and the values are the quantiles.
- If  $q$  is a float, a Series will be returned where the index is the columns of self and the values are the quantiles.

**See also:**

[`pandas.core.window.Rolling.quantile`](#)

## Examples

```
>>> df = pd.DataFrame(np.array([[1, 1], [2, 10], [3, 100], [4, 100]]),
                      columns=['a', 'b'])
>>> df.quantile(.1)
a    1.3
b    3.7
dtype: float64
>>> df.quantile([.1, .5])
      a    b
0.1  1.3  3.7
0.5  2.5 55.0
```

Specifying `numeric_only=False` will also compute the quantile of datetime and timedelta data.

```
>>> df = pd.DataFrame({'A': [1, 2],
                      'B': [pd.Timestamp('2010'),
                           pd.Timestamp('2011')],
                      'C': [pd.Timedelta('1 days'),
                           pd.Timedelta('2 days')]})
>>> df.quantile(0.5, numeric_only=False)
A    1.5
```

(continues on next page)

(continued from previous page)

```
B    2010-07-02 12:00:00
C         1 days 12:00:00
Name: 0.5, dtype: object
```

### 34.17.3.49 pandas.core.groupby.DataFrameGroupBy.rank

DataFrameGroupBy.**rank** (*method='average', ascending=True, na\_option='keep', pct=False, axis=0*)

Provides the rank of values within each group.

**Parameters method** : {'average', 'min', 'max', 'first', 'dense'}, default 'average'

- average: average rank of group
- min: lowest rank in group
- max: highest rank in group
- first: ranks assigned in order they appear in the array
- dense: like 'min', but rank always increases by 1 between groups

**ascending** : boolean, default True

False for ranks by high (1) to low (N)

**na\_option** : {'keep', 'top', 'bottom'}, default 'keep'

- keep: leave NA values where they are
- top: smallest rank if ascending
- bottom: smallest rank if descending

**pct** : boolean, default False

Compute percentage rank of data within each group

**axis** : int, default 0

The axis of the object over which to compute the rank.

**Returns**

—

**DataFrame with ranking of values within each group**

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.50 pandas.core.groupby.DataFrameGroupBy.resample

DataFrameGroupBy.**resample** (*rule, \*args, \*\*kwargs*)

Provide resampling when using a TimeGrouper Return a new grouper with our resampler appended

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*



### 34.17.3.51 pandas.core.groupby.DataFrameGroupBy.shift

DataFrameGroupBy.**shift** (*periods=1, freq=None, axis=0*)

Shift each group by periods observations

**Parameters** **periods** : integer, default 1

number of periods to shift

**freq** [frequency string]

**axis** [axis to shift, default 0]

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.52 pandas.core.groupby.DataFrameGroupBy.size

DataFrameGroupBy.**size** ()

Compute group sizes

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

### 34.17.3.53 pandas.core.groupby.DataFrameGroupBy.skew

DataFrameGroupBy.**skew**

Return unbiased skew over requested axis Normalized by N-1

**Parameters**

**axis** [{index (0), columns (1)}]

**skipna** : boolean, default True

Exclude NA/null values when computing the result.

**level** : int or level name, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a Series

**numeric\_only** : boolean, default None

Include only float, int, boolean columns. If None, will attempt to use everything, then use only numeric data. Not implemented for Series.

**Returns**

**skew** [Series or DataFrame (if level specified)]

### 34.17.3.54 pandas.core.groupby.DataFrameGroupBy.take

DataFrameGroupBy.**take**

Return the elements in the given *positional* indices along an axis.

This means that we are not indexing according to actual values in the index attribute of the object. We are indexing according to the actual position of the element in the object.

**Parameters** `indices` : array-like

An array of ints indicating which positions to take.

**axis** : {0 or 'index', 1 or 'columns', None}, default 0

The axis on which to select elements. 0 means that we are selecting rows, 1 means that we are selecting columns.

**convert** : bool, default TrueWhether to convert negative indices into positive ones. For example, `-1` would map to the `len(axis) - 1`. The conversions are similar to the behavior of indexing a regular Python list.

Deprecated since version 0.21.0: In the future, negative indices will always be converted.

**is\_copy** : bool, default True

Whether to return a copy of the original object or not.

**\*\*kwargs**For compatibility with `numpy.take()`. Has no effect on the output.**Returns** `taken` : type of caller

An array-like containing the elements taken from the object.

**See also:****DataFrame.loc** Select a subset of a DataFrame by labels.**DataFrame.iloc** Select a subset of a DataFrame by positions.**numpy.take** Take elements from an array along an axis.

## Examples

```
>>> df = pd.DataFrame([('falcon', 'bird', 389.0),
...                     ('parrot', 'bird', 24.0),
...                     ('lion', 'mammal', 80.5),
...                     ('monkey', 'mammal', np.nan)],
...                     columns=['name', 'class', 'max_speed'],
...                     index=[0, 2, 3, 1])
>>> df
   name  class  max_speed
0  falcon   bird    389.0
2  parrot   bird     24.0
3   lion  mammal     80.5
1  monkey  mammal      NaN
```

Take elements at positions 0 and 3 along the axis 0 (default).

Note how the actual indices selected (0 and 1) do not correspond to our selected indices 0 and 3. That's because we are selecting the 0th and 3rd rows, not rows whose indices equal 0 and 3.

```
>>> df.take([0, 3])
   name  class  max_speed
0  falcon   bird    389.0
1  monkey  mammal      NaN
```

Take elements at indices 1 and 2 along the axis 1 (column selection).

```
>>> df.take([1, 2], axis=1)
      class  max_speed
0     bird      389.0
2     bird       24.0
3  mammal      80.5
1  mammal       NaN
```

We may take elements using negative integers for positive indices, starting from the end of the object, just like with Python lists.

```
>>> df.take([-1, -2])
      name  class  max_speed
1  monkey  mammal       NaN
3    lion  mammal      80.5
```

### 34.17.3.55 pandas.core.groupby.DataFrameGroupBy.tshift

DataFrameGroupBy.**tshift**

Shift the time index, using the index's frequency if available.

**Parameters** **periods** : int

Number of periods to move, can be positive or negative

**freq** : DateOffset, timedelta, or time rule string, default None

Increment to use from the tseries module or time rule (e.g. 'EOM')

**axis** : int or basestring

Corresponds to the axis that contains the Index

**Returns**

**shifted** [NDFrame]

#### Notes

If freq is not specified then tries to use the freq or inferred\_freq attributes of the index. If neither of those attributes exist, a ValueError is thrown

The following methods are available only for SeriesGroupBy objects.

<i>SeriesGroupBy.nlargest</i>	Return the largest <i>n</i> elements.
<i>SeriesGroupBy.nsmallest</i>	Return the smallest <i>n</i> elements.
<i>SeriesGroupBy.nunique([dropna])</i>	Returns number of unique elements in the group
<i>SeriesGroupBy.unique</i>	Return unique values of Series object.
<i>SeriesGroupBy.value_counts([normalize, ...])</i>	
<i>SeriesGroupBy.is_monotonic_increasing</i>	Return boolean if values in the object are monotonic_increasing
<i>SeriesGroupBy.is_monotonic_decreasing</i>	Return boolean if values in the object are monotonic_decreasing

### 34.17.3.56 pandas.core.groupby.SeriesGroupBy.nlargest

**SeriesGroupBy.nlargest**

Return the largest  $n$  elements.

**Parameters** **n** : int

Return this many descending sorted values

**keep** : {‘first’, ‘last’}, default ‘first’

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

**Returns** **top\_n** : Series

The  $n$  largest values in the Series, in sorted order

**See also:**

`Series.nsmallest`

#### Notes

Faster than `.sort_values(ascending=False).head(n)` for small  $n$  relative to the size of the Series object.

#### Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nlargest(10)  # only sorts up to the N requested
219921    4.644710
82124     4.608745
421689    4.564644
425277    4.447014
718691    4.414137
43154     4.403520
283187    4.313922
595519    4.273635
503969    4.250236
121637    4.240952
dtype: float64
```

### 34.17.3.57 pandas.core.groupby.SeriesGroupBy.nsmallest

**SeriesGroupBy.nsmallest**

Return the smallest  $n$  elements.

**Parameters** **n** : int

Return this many ascending sorted values

**keep** : {‘first’, ‘last’}, default ‘first’

Where there are duplicate values: - `first` : take the first occurrence. - `last` : take the last occurrence.

**Returns** `bottom_n` : Series

The `n` smallest values in the Series, in sorted order

**See also:**

`Series.nlargest`

## Notes

Faster than `.sort_values().head(n)` for small `n` relative to the size of the Series object.

## Examples

```
>>> import pandas as pd
>>> import numpy as np
>>> s = pd.Series(np.random.randn(10**6))
>>> s.nsmallest(10)  # only sorts up to the N requested
288532    -4.954580
732345    -4.835960
64803     -4.812550
446457    -4.609998
501225    -4.483945
669476    -4.472935
973615    -4.401699
621279    -4.355126
773916    -4.347355
359919    -4.331927
dtype: float64
```

### 34.17.3.58 pandas.core.groupby.SeriesGroupBy.nunique

`SeriesGroupBy.nunique` (`dropna=True`)

Returns number of unique elements in the group

### 34.17.3.59 pandas.core.groupby.SeriesGroupBy.unique

`SeriesGroupBy.unique`

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

**Returns** `ndarray` or `Categorical`

The unique values returned as a NumPy array. In case of categorical data type, returned as a Categorical.

**See also:**

`pandas.unique` top-level unique method for any 1-d array-like object.

`Index.unique` return Index with unique values from an Index object.

## Examples

```
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...           for _ in range(3)]).unique()
array([Timestamp('2016-01-01 00:00:00-0500', tz='US/Eastern')],
      dtype=object)
```

An unordered Categorical will return categories in the order of appearance.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
[b, a, c]
Categories (3, object): [b, a, c]
```

An ordered Categorical preserves the category ordering.

```
>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
...                           ordered=True)).unique()
[b, a, c]
Categories (3, object): [a < b < c]
```

### 34.17.3.60 pandas.core.groupby.SeriesGroupBy.value\_counts

`SeriesGroupBy.value_counts` (*normalize=False*, *sort=True*, *ascending=False*, *bins=None*, *dropna=True*)

### 34.17.3.61 pandas.core.groupby.SeriesGroupBy.is\_monotonic\_increasing

`SeriesGroupBy.is_monotonic_increasing`

Return boolean if values in the object are monotonic\_increasing

New in version 0.19.0.

#### Returns

`is_monotonic` [boolean]

### 34.17.3.62 pandas.core.groupby.SeriesGroupBy.is\_monotonic\_decreasing

`SeriesGroupBy.is_monotonic_decreasing`

Return boolean if values in the object are monotonic\_decreasing

New in version 0.19.0.

#### Returns

`is_monotonic_decreasing` [boolean]

The following methods are available only for `DataFrameGroupBy` objects.

---

<code>DataFrameGroupBy.corrwith</code>	Compute pairwise correlation between rows or columns of two DataFrame objects.
<code>DataFrameGroupBy.boxplot([subplots, column, ...])</code>	Make box plots from DataFrameGroupBy data.

---

### 34.17.3.63 pandas.core.groupby.DataFrameGroupBy.corrwith

DataFrameGroupBy.**corrwith**

Compute pairwise correlation between rows or columns of two DataFrame objects.

#### Parameters

**other** [DataFrame, Series]

**axis** : {0 or 'index', 1 or 'columns'}, default 0

0 or 'index' to compute column-wise, 1 or 'columns' for row-wise

**drop** : boolean, default False

Drop missing indices from result, default returns union of all

#### Returns

**correls** [Series]

### 34.17.3.64 pandas.core.groupby.DataFrameGroupBy.boxplot

DataFrameGroupBy.**boxplot** (*subplots=True, column=None, fontsize=None, rot=0, grid=True, ax=None, figsize=None, layout=None, \*\*kwargs*)

Make box plots from DataFrameGroupBy data.

#### Parameters

**grouped** [Grouped DataFrame]

**subplots** :

- False - no subplots will be used
- True - create a subplot for each group

**column** : column name or list of names, or vector

Can be any valid input to groupby

**fontsize** [int or string]

**rot** [label rotation angle]

**grid** [Setting this to True will show the grid]

**ax** [Matplotlib axis object, default None]

**figsize** [A tuple (width, height) in inches]

**layout** : tuple (optional)

(rows, columns) for the layout of the plot

**\*\*kwargs** : Keyword Arguments

All other plotting keyword arguments to be passed to matplotlib's boxplot function

#### Returns

dict of key/value = group key/DataFrame.boxplot return value

or DataFrame.boxplot return value in case subplots=figures=False

#### Examples

```
>>> import pandas
>>> import numpy as np
>>> import itertools
>>>
>>> tuples = [t for t in itertools.product(range(1000), range(4))]
>>> index = pandas.MultiIndex.from_tuples(tuples, names=['lvl0', 'lvl1'])
>>> data = np.random.randn(len(index), 4)
>>> df = pandas.DataFrame(data, columns=list('ABCD'), index=index)
>>>
>>> grouped = df.groupby(level='lvl1')
>>> boxplot_frame_groupby(grouped)
>>>
>>> grouped = df.unstack(level='lvl1').groupby(level=0, axis=1)
>>> boxplot_frame_groupby(grouped, subplots=False)
```

## 34.18 Resampling

Resampler objects are returned by resample calls: `pandas.DataFrame.resample()`, `pandas.Series.resample()`.

### 34.18.1 Indexing, iteration

<code>Resampler.__iter__()</code>	Groupby iterator
<code>Resampler.groups</code>	dict {group name -> group labels}
<code>Resampler.indices</code>	dict {group name -> group indices}
<code>Resampler.get_group(name[, obj])</code>	Constructs NDFrame from group with provided name

#### 34.18.1.1 pandas.core.resample.Resampler.\_\_iter\_\_

`Resampler.__iter__()`

Groupby iterator

#### Returns

Generator yielding sequence of (name, subsetted object)  
for each group

#### 34.18.1.2 pandas.core.resample.Resampler.groups

`Resampler.groups`

dict {group name -> group labels}



### 34.18.1.3 pandas.core.resample.Resampler.indices

`Resampler.indices`

dict {group name -> group indices}

### 34.18.1.4 pandas.core.resample.Resampler.get\_group

`Resampler.get_group(name, obj=None)`

Constructs NDFrame from group with provided name

**Parameters** `name` : object

the name of the group to get as a DataFrame

`obj` : NDFrame, default None

the NDFrame to take the DataFrame out of. If it is None, the object groupby was called on will be used

**Returns**

`group` [type of obj]

## 34.18.2 Function application

<code>Resampler.apply(arg, *args, **kwargs)</code>	Aggregate using one or more operations over the specified axis.
<code>Resampler.aggregate(arg, *args, **kwargs)</code>	Aggregate using one or more operations over the specified axis.
<code>Resampler.transform(arg, *args, **kwargs)</code>	Call function producing a like-indexed Series on each group and return a Series with the transformed values
<code>Resampler.pipe(func, *args, **kwargs)</code>	Apply a function <code>func</code> with arguments to this Resampler object and return the function's result.

### 34.18.2.1 pandas.core.resample.Resampler.apply

`Resampler.apply(arg, *args, **kwargs)`

Aggregate using one or more operations over the specified axis.

**Parameters** `func` : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**\*args**

Positional arguments to pass to `func`.

**\*\*kwargs**Keyword arguments to pass to *func*.**Returns****aggregated** [DataFrame]**See also:**

pandas.DataFrame.groupby.aggregate, pandas.DataFrame.resample.transform,  
[pandas.DataFrame.aggregate](#)

**Notes**

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

**Examples**

```
>>> s = Series([1,2,3,4,5],
               index=pd.date_range('20130101',
                                   periods=5,freq='s'))
2013-01-01 00:00:00    1
2013-01-01 00:00:01    2
2013-01-01 00:00:02    3
2013-01-01 00:00:03    4
2013-01-01 00:00:04    5
Freq: S, dtype: int64
```

```
>>> r = s.resample('2s')
DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left,
                        label=left, convention=start, base=0]
```

```
>>> r.agg(np.sum)
2013-01-01 00:00:00    3
2013-01-01 00:00:02    7
2013-01-01 00:00:04    5
Freq: 2S, dtype: int64
```

```
>>> r.agg(['sum', 'mean', 'max'])
               sum  mean  max
2013-01-01 00:00:00    3   1.5    2
2013-01-01 00:00:02    7   3.5    4
2013-01-01 00:00:04    5   5.0    5
```

```
>>> r.agg({'result' : lambda x: x.mean() / x.std(),
          'total' : np.sum})
               total  result
2013-01-01 00:00:00    3  2.121320
2013-01-01 00:00:02    7  4.949747
2013-01-01 00:00:04    5         NaN
```

### 34.18.2.2 pandas.core.resample.Resampler.aggregate

`Resampler.aggregate` (*arg*, \**args*, \*\**kwargs*)

Aggregate using one or more operations over the specified axis.

**Parameters** *func* : function, string, dictionary, or list of string/functions

Function to use for aggregating the data. If a function, must either work when passed a DataFrame or when passed to DataFrame.apply. For a DataFrame, can pass a dict, if the keys are DataFrame column names.

Accepted combinations are:

- string function name.
- function.
- list of functions.
- dict of column names -> functions (or list of functions).

**\*args**

Positional arguments to pass to *func*.

**\*\*kwargs**

Keyword arguments to pass to *func*.

**Returns**

**aggregated** [DataFrame]

**See also:**

`pandas.DataFrame.groupby.aggregate`, `pandas.DataFrame.resample.transform`,  
[\*pandas.DataFrame.aggregate\*](#)

#### Notes

*agg* is an alias for *aggregate*. Use the alias.

A passed user-defined-function will be passed a Series for evaluation.

#### Examples

```
>>> s = Series([1,2,3,4,5],
               index=pd.date_range('20130101',
                                   periods=5,freq='s'))
2013-01-01 00:00:00    1
2013-01-01 00:00:01    2
2013-01-01 00:00:02    3
2013-01-01 00:00:03    4
2013-01-01 00:00:04    5
Freq: S, dtype: int64
```

```
>>> r = s.resample('2s')
DatetimeIndexResampler [freq=<2 * Seconds>, axis=0, closed=left,
                        label=left, convention=start, base=0]
```

```
>>> r.agg(np.sum)
2013-01-01 00:00:00    3
2013-01-01 00:00:02    7
2013-01-01 00:00:04    5
Freq: 2S, dtype: int64
```

```
>>> r.agg(['sum', 'mean', 'max'])
              sum  mean  max
2013-01-01 00:00:00    3   1.5    2
2013-01-01 00:00:02    7   3.5    4
2013-01-01 00:00:04    5   5.0    5
```

```
>>> r.agg({'result' : lambda x: x.mean() / x.std(),
          'total' : np.sum})
              total  result
2013-01-01 00:00:00    3  2.121320
2013-01-01 00:00:02    7  4.949747
2013-01-01 00:00:04    5         NaN
```

### 34.18.2.3 pandas.core.resample.Resampler.transform

`Resampler.transform(arg, *args, **kwargs)`

Call function producing a like-indexed Series on each group and return a Series with the transformed values

**Parameters** `func` : function

To apply to each group. Should return a Series with the same index

**Returns**

**transformed** [Series]

#### Examples

```
>>> resampled.transform(lambda x: (x - x.mean()) / x.std())
```

### 34.18.2.4 pandas.core.resample.Resampler.pipe

`Resampler.pipe(func, *args, **kwargs)`

Apply a function `func` with arguments to this Resampler object and return the function's result.

New in version 0.23.0.

Use `.pipe` when you want to improve readability by chaining together functions that expect Series, DataFrames, GroupBy or Resampler objects. Instead of writing

```
>>> h(g(f(df.groupby('group')), arg1=a), arg2=b, arg3=c)
```

You can write

```
>>> (df.groupby('group')
...   .pipe(f)
...   .pipe(g, arg1=a)
...   .pipe(h, arg2=b, arg3=c))
```

which is much more readable.

**Parameters** `func` : callable or tuple of (callable, string)

Function to apply to this Resampler object or, alternatively, a (callable, data\_keyword) tuple where data\_keyword is a string indicating the keyword of callable that expects the Resampler object.

**args** : iterable, optional

positional arguments passed into `func`.

**kwargs** : dict, optional

a dictionary of keyword arguments passed into `func`.

**Returns**

**object** [the return type of `func`.]

**See also:**

*[pandas.Series.pipe](#)* Apply a function with arguments to a series

*[pandas.DataFrame.pipe](#)* Apply a function with arguments to a dataframe

*[apply](#)* Apply function to each group instead of to the full Resampler object.

## Notes

See more [here](#)

## Examples

```
>>> df = pd.DataFrame({'A': [1, 2, 3, 4]},
...                    index=pd.date_range('2012-08-02', periods=4))
>>> df
           A
2012-08-02  1
2012-08-03  2
2012-08-04  3
2012-08-05  4
```

To get the difference between each 2-day period's maximum and minimum value in one pass, you can do

```
>>> df.resample('2D').pipe(lambda x: x.max() - x.min())
           A
2012-08-02  1
2012-08-04  1
```

### 34.18.3 Upsampling

<i><a href="#">Resampler.ffill([limit])</a></i>	Forward fill the values
<i><a href="#">Resampler.bfill([limit])</a></i>	Backward fill the new missing values in the resampled data.

Continued on next page

Table 172 – continued from previous page

<code>Resampler.bfill([limit])</code>	Backward fill the new missing values in the resampled data.
<code>Resampler.pad([limit])</code>	Forward fill the values
<code>Resampler.nearest([limit])</code>	Fill values with nearest neighbor starting from center
<code>Resampler.fillna(method[, limit])</code>	Fill missing values introduced by upsampling.
<code>Resampler.asfreq([fill_value])</code>	return the values at the new freq, essentially a reindex
<code>Resampler.interpolate([method, axis, limit, ...])</code>	Interpolate values according to different methods.

### 34.18.3.1 pandas.core.resample.Resampler.ffill

`Resampler.ffi11` (*limit=None*)

Forward fill the values

**Parameters** `limit` : integer, optional

limit of how many values to fill

**Returns**

an upsampled Series

**See also:**

`Series.fillna`, `DataFrame.fillna`

### 34.18.3.2 pandas.core.resample.Resampler.backfill

`Resampler.backfill` (*limit=None*)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [R30]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

**Parameters** `limit` : integer, optional

Limit of how many values to fill.

**Returns** Series, DataFrame

An upsampled Series or DataFrame with backward filled NaN values.

**See also:**

`bfill` Alias of backfill.

`fillna` Fill NaN values using the specified method, which can be ‘backfill’.

`nearest` Fill NaN values with nearest neighbor starting from center.

`pad` Forward fill NaN values.

`pandas.Series.fillna` Fill NaN values in the Series using the specified method, which can be ‘backfill’.

`pandas.DataFrame.fillna` Fill NaN values in the DataFrame using the specified method, which can be ‘backfill’.

## References

[R30]

## Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...                 index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

```
>>> s.resample('30min').backfill()
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').backfill(limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

Resampling a DataFrame that has missing values:

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                     index=pd.date_range('20180101', periods=3,
...                                           freq='h'))
>>> df
              a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 01:00:00  NaN  3
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('30min').backfill()
              a  b
2018-01-01 00:00:00  2.0  1
2018-01-01 00:30:00  NaN  3
2018-01-01 01:00:00  NaN  3
2018-01-01 01:30:00  6.0  5
2018-01-01 02:00:00  6.0  5
```

```
>>> df.resample('15min').backfill(limit=2)
              a      b
2018-01-01 00:00:00  2.0  1.0
2018-01-01 00:15:00  NaN  NaN
2018-01-01 00:30:00  NaN  3.0
2018-01-01 00:45:00  NaN  3.0
2018-01-01 01:00:00  NaN  3.0
2018-01-01 01:15:00  NaN  NaN
2018-01-01 01:30:00  6.0  5.0
2018-01-01 01:45:00  6.0  5.0
2018-01-01 02:00:00  6.0  5.0
```

### 34.18.3.3 pandas.core.resample.Resampler.bfill

`Resampler.bfill` (*limit=None*)

Backward fill the new missing values in the resampled data.

In statistics, imputation is the process of replacing missing data with substituted values [R31]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency). The backward fill will replace NaN values that appeared in the resampled data with the next value in the original sequence. Missing values that existed in the original data will not be modified.

**Parameters** `limit` : integer, optional

Limit of how many values to fill.

**Returns** `Series, DataFrame`

An upsampled Series or DataFrame with backward filled NaN values.

**See also:**

**bfill** Alias of backfill.

**fillna** Fill NaN values using the specified method, which can be ‘backfill’.

**nearest** Fill NaN values with nearest neighbor starting from center.

**pad** Forward fill NaN values.

**pandas.Series.fillna** Fill NaN values in the Series using the specified method, which can be ‘backfill’.

**pandas.DataFrame.fillna** Fill NaN values in the DataFrame using the specified method, which can be ‘backfill’.

## References

[R31]

## Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
```

(continues on next page)



(continued from previous page)

```

2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64

```

```

>>> s.resample('30min').backfill()
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64

```

```

>>> s.resample('15min').backfill(limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64

```

Resampling a DataFrame that has missing values:

```

>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                     index=pd.date_range('20180101', periods=3,
...                     freq='h'))
>>> df

```

	a	b
2018-01-01 00:00:00	2.0	1
2018-01-01 01:00:00	NaN	3
2018-01-01 02:00:00	6.0	5

```

>>> df.resample('30min').backfill()

```

	a	b
2018-01-01 00:00:00	2.0	1
2018-01-01 00:30:00	NaN	3
2018-01-01 01:00:00	NaN	3
2018-01-01 01:30:00	6.0	5
2018-01-01 02:00:00	6.0	5

```

>>> df.resample('15min').backfill(limit=2)

```

	a	b
2018-01-01 00:00:00	2.0	1.0
2018-01-01 00:15:00	NaN	NaN
2018-01-01 00:30:00	NaN	3.0
2018-01-01 00:45:00	NaN	3.0
2018-01-01 01:00:00	NaN	3.0
2018-01-01 01:15:00	NaN	NaN
2018-01-01 01:30:00	6.0	5.0
2018-01-01 01:45:00	6.0	5.0
2018-01-01 02:00:00	6.0	5.0

#### 34.18.3.4 `pandas.core.resample.Resampler.pad`

`Resampler.pad` (*limit=None*)

Forward fill the values

**Parameters** `limit` : integer, optional

limit of how many values to fill

**Returns**

an upsampled Series

**See also:**

`Series.fillna`, `DataFrame.fillna`

#### 34.18.3.5 `pandas.core.resample.Resampler.nearest`

`Resampler.nearest` (*limit=None*)

Fill values with nearest neighbor starting from center

**Parameters** `limit` : integer, optional

limit of how many values to fill

New in version 0.21.0.

**Returns**

an upsampled Series

**See also:**

`Series.fillna`, `DataFrame.fillna`

#### 34.18.3.6 `pandas.core.resample.Resampler.fillna`

`Resampler.fillna` (*method*, *limit=None*)

Fill missing values introduced by upsampling.

In statistics, imputation is the process of replacing missing data with substituted values [R32]. When resampling data, missing values may appear (e.g., when the resampling frequency is higher than the original frequency).

Missing values that existed in the original data will not be modified.

**Parameters** `method` : {'pad', 'backfill', 'ffill', 'bfill', 'nearest'}

Method to use for filling holes in resampled data

- 'pad' or 'ffill': use previous valid observation to fill gap (forward fill).
- 'backfill' or 'bfill': use next valid observation to fill gap.
- 'nearest': use nearest valid observation to fill gap.

**limit** : integer, optional

Limit of how many consecutive missing values to fill.

**Returns** Series or DataFrame

An upsampled Series or DataFrame with missing values filled.

**See also:**

**backfill** Backward fill NaN values in the resampled data.

**pad** Forward fill NaN values in the resampled data.

**nearest** Fill NaN values in the resampled data with nearest neighbor starting from center.

**interpolate** Fill NaN values using interpolation.

**pandas.Series.fillna** Fill NaN values in the Series using the specified method, which can be 'bfill' and 'ffill'.

**pandas.DataFrame.fillna** Fill NaN values in the DataFrame using the specified method, which can be 'bfill' and 'ffill'.

## References

[R32]

## Examples

Resampling a Series:

```
>>> s = pd.Series([1, 2, 3],
...               index=pd.date_range('20180101', periods=3, freq='h'))
>>> s
2018-01-01 00:00:00    1
2018-01-01 01:00:00    2
2018-01-01 02:00:00    3
Freq: H, dtype: int64
```

Without filling the missing values you get:

```
>>> s.resample("30min").asfreq()
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    2.0
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> s.resample('30min').fillna("backfill")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('15min').fillna("backfill", limit=2)
2018-01-01 00:00:00    1.0
2018-01-01 00:15:00    NaN
2018-01-01 00:30:00    2.0
2018-01-01 00:45:00    2.0
2018-01-01 01:00:00    2.0
2018-01-01 01:15:00    NaN
2018-01-01 01:30:00    3.0
```

(continues on next page)

(continued from previous page)

```
2018-01-01 01:45:00    3.0
2018-01-01 02:00:00    3.0
Freq: 15T, dtype: float64
```

```
>>> s.resample('30min').fillna("pad")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    1
2018-01-01 01:00:00    2
2018-01-01 01:30:00    2
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

```
>>> s.resample('30min').fillna("nearest")
2018-01-01 00:00:00    1
2018-01-01 00:30:00    2
2018-01-01 01:00:00    2
2018-01-01 01:30:00    3
2018-01-01 02:00:00    3
Freq: 30T, dtype: int64
```

Missing values present before the upsampling are not affected.

```
>>> sm = pd.Series([1, None, 3],
...                 index=pd.date_range('20180101', periods=3, freq='h'))
>>> sm
2018-01-01 00:00:00    1.0
2018-01-01 01:00:00    NaN
2018-01-01 02:00:00    3.0
Freq: H, dtype: float64
```

```
>>> sm.resample('30min').fillna('backfill')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> sm.resample('30min').fillna('pad')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    1.0
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    NaN
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

```
>>> sm.resample('30min').fillna('nearest')
2018-01-01 00:00:00    1.0
2018-01-01 00:30:00    NaN
2018-01-01 01:00:00    NaN
2018-01-01 01:30:00    3.0
2018-01-01 02:00:00    3.0
Freq: 30T, dtype: float64
```

DataFrame resampling is done column-wise. All the same options are available.

```
>>> df = pd.DataFrame({'a': [2, np.nan, 6], 'b': [1, 3, 5]},
...                     index=pd.date_range('20180101', periods=3,
...                     freq='h'))
>>> df
```

	a	b
2018-01-01 00:00:00	2.0	1
2018-01-01 01:00:00	NaN	3
2018-01-01 02:00:00	6.0	5

```
>>> df.resample('30min').fillna("bfill")
```

	a	b
2018-01-01 00:00:00	2.0	1
2018-01-01 00:30:00	NaN	3
2018-01-01 01:00:00	NaN	3
2018-01-01 01:30:00	6.0	5
2018-01-01 02:00:00	6.0	5

### 34.18.3.7 pandas.core.resample.Resampler.asfreq

`Resampler.asfreq` (*fill\_value=None*)

return the values at the new freq, essentially a reindex

**Parameters** *fill\_value*: scalar, optional

Value to use for missing values, applied during upsampling (note this does not fill NaNs that already were present).

New in version 0.20.0.

**See also:**

`Series.asfreq`, `DataFrame.asfreq`

### 34.18.3.8 pandas.core.resample.Resampler.interpolate

`Resampler.interpolate` (*method='linear', axis=0, limit=None, inplace=False, limit\_direction='forward', limit\_area=None, downcast=None, \*\*kwargs*)

Interpolate values according to different methods.

New in version 0.18.1.

Please note that only `method='linear'` is supported for DataFrames/Series with a MultiIndex.

**Parameters** *method*: {'linear', 'time', 'index', 'values', 'nearest', 'zero',

'slinear', 'quadratic', 'cubic', 'barycentric', 'krogh', 'polynomial', 'spline', 'piecewise\_polynomial', 'from\_derivatives', 'pchip', 'akima'}

- 'linear': ignore the index and treat the values as equally spaced. This is the only method supported on MultiIndexes. default
- 'time': interpolation works on daily and higher resolution data to interpolate given length of interval
- 'index', 'values': use the actual numerical values of the index
- 'nearest', 'zero', 'slinear', 'quadratic', 'cubic', 'barycentric', 'polynomial' is passed to `scipy.interpolate.interpld`. Both 'polynomial' and 'spline' require that you also specify an *order* (int), e.g.

`df.interpolate(method='polynomial', order=4)`. These use the actual numerical values of the index.

- 'krogh', 'piecewise\_polynomial', 'spline', 'pchip' and 'akima' are all wrappers around the scipy interpolation methods of similar names. These use the actual numerical values of the index. For more information on their behavior, see the [scipy documentation](#) and [tutorial documentation](#)
- 'from\_derivatives' refers to `BPoly.from_derivatives` which replaces 'piecewise\_polynomial' interpolation method in scipy 0.18

New in version 0.18.1: Added support for the 'akima' method Added interpolate method 'from\_derivatives' which replaces 'piecewise\_polynomial' in scipy 0.18; backwards-compatible with scipy < 0.18

**axis** : {0, 1}, default 0

- 0: fill column-by-column
- 1: fill row-by-row

**limit** : int, default None.

Maximum number of consecutive NaNs to fill. Must be greater than 0.

**limit\_direction** [{ 'forward', 'backward', 'both' }, default 'forward']

**limit\_area** : { 'inside', 'outside' }, default None

- None: (default) no fill restriction
- 'inside' Only fill NaNs surrounded by valid values (interpolate).
- 'outside' Only fill NaNs outside valid values (extrapolate).

If limit is specified, consecutive NaNs will be filled in this direction.

New in version 0.21.0.

**inplace** : bool, default False

Update the NDFrame in place if possible.

**downcast** : optional, 'infer' or None, defaults to None

Downcast dtypes if possible.

**kwargs** [keyword arguments to pass on to the interpolating function.]

### Returns

**Series or DataFrame of same shape interpolated at the NaNs**

See also:

`reindex`, `replace`, `fillna`

### Examples

Filling in NaNs

```

>>> s = pd.Series([0, 1, np.nan, 3])
>>> s.interpolate()
0    0
1    1
2    2
3    3
dtype: float64

```

### 34.18.4 Computations / Descriptive Stats

<i>Resampler.count</i> ([_method])	Compute count of group, excluding missing values
<i>Resampler.nunique</i> ([_method])	Returns number of unique elements in the group
<i>Resampler.first</i> ([_method])	Compute first of group values
<i>Resampler.last</i> ([_method])	Compute last of group values
<i>Resampler.max</i> ([_method])	Compute max of group values
<i>Resampler.mean</i> ([_method])	Compute mean of groups, excluding missing values
<i>Resampler.median</i> ([_method])	Compute median of groups, excluding missing values
<i>Resampler.min</i> ([_method])	Compute min of group values
<i>Resampler.ohlc</i> ([_method])	Compute sum of values, excluding missing values For multiple groupings, the result index will be a MultiIndex
<i>Resampler.prod</i> ([_method, min_count])	Compute prod of group values
<i>Resampler.size</i> ()	Compute group sizes
<i>Resampler.sem</i> ([_method])	Compute standard error of the mean of groups, excluding missing values
<i>Resampler.std</i> ([ddof])	Compute standard deviation of groups, excluding missing values
<i>Resampler.sum</i> ([_method, min_count])	Compute sum of group values
<i>Resampler.var</i> ([ddof])	Compute variance of groups, excluding missing values

#### 34.18.4.1 pandas.core.resample.Resampler.count

`Resampler.count` (*\_method*='count')

Compute count of group, excluding missing values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.2 pandas.core.resample.Resampler.nunique

`Resampler.nunique` (*\_method*='nunique')

Returns number of unique elements in the group

#### 34.18.4.3 pandas.core.resample.Resampler.first

`Resampler.first` (*\_method*='first', \*args, \*\*kwargs)

Compute first of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.4 pandas.core.resample.Resampler.last

`Resampler.last` (*\_method='last', \*args, \*\*kwargs*)  
Compute last of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.5 pandas.core.resample.Resampler.max

`Resampler.max` (*\_method='max', \*args, \*\*kwargs*)  
Compute max of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.6 pandas.core.resample.Resampler.mean

`Resampler.mean` (*\_method='mean', \*args, \*\*kwargs*)  
Compute mean of groups, excluding missing values  
For multiple groupings, the result index will be a MultiIndex

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.7 pandas.core.resample.Resampler.median

`Resampler.median` (*\_method='median', \*args, \*\*kwargs*)  
Compute median of groups, excluding missing values  
For multiple groupings, the result index will be a MultiIndex

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.8 pandas.core.resample.Resampler.min

`Resampler.min` (*\_method='min', \*args, \*\*kwargs*)  
Compute min of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.9 pandas.core.resample.Resampler.ohlc

`Resampler.ohlc` (*\_method='ohlc', \*args, \*\*kwargs*)  
Compute sum of values, excluding missing values For multiple groupings, the result index will be a MultiIndex

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*



#### 34.18.4.10 pandas.core.resample.Resampler.prod

`Resampler.prod(_method='prod', min_count=0, *args, **kwargs)`  
Compute prod of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.11 pandas.core.resample.Resampler.size

`Resampler.size()`  
Compute group sizes

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.12 pandas.core.resample.Resampler.sem

`Resampler.sem(_method='sem', *args, **kwargs)`  
Compute standard error of the mean of groups, excluding missing values  
For multiple groupings, the result index will be a MultiIndex

**Parameters** **ddof**: integer, default 1  
degrees of freedom

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.13 pandas.core.resample.Resampler.std

`Resampler.std(ddof=1, *args, **kwargs)`  
Compute standard deviation of groups, excluding missing values

**Parameters**  
**ddof** [integer, default 1]  
degrees of freedom

#### 34.18.4.14 pandas.core.resample.Resampler.sum

`Resampler.sum(_method='sum', min_count=0, *args, **kwargs)`  
Compute sum of group values

**See also:**

*pandas.Series.groupby, pandas.DataFrame.groupby, pandas.Panel.groupby*

#### 34.18.4.15 pandas.core.resample.Resampler.var

`Resampler.var(ddof=1, *args, **kwargs)`

Compute variance of groups, excluding missing values

##### Parameters

**ddof** [integer, default 1]

**degrees of freedom**

### 34.19 Style

Styler objects are returned by `pandas.DataFrame.style`.

#### 34.19.1 Styler Constructor

---

<code>Styler(data[, precision, table_styles, ...])</code>	Helps style a DataFrame or Series according to the data with HTML and CSS.
<code>Styler.from_custom_template(searchpath, name)</code>	Factory function for creating a subclass of Styler with a custom template and Jinja environment.

---

##### 34.19.1.1 pandas.io.formats.style.Styler

**class** `pandas.io.formats.style.Styler` (*data, precision=None, table\_styles=None, uuid=None, caption=None, table\_attributes=None*)

Helps style a DataFrame or Series according to the data with HTML and CSS.

##### Parameters

**data: Series or DataFrame**

**precision: int**

precision to round floats to, defaults to `pd.options.display.precision`

**table\_styles: list-like, default None**

list of {selector: (attr, value)} dicts; see Notes

**uuid: str, default None**

a unique identifier to avoid CSS collisions; generated automatically

**caption: str, default None**

caption to attach to the table

##### See also:

`pandas.DataFrame.style`

##### Notes

Most styling will be done by passing style functions into `Styler.apply` or `Styler.applymap`. Style functions should return values with strings containing CSS `'attr: value'` that will be applied to the indicated cells.

If using in the Jupyter notebook, Styler has defined a `_repr_html_` to automatically render itself. Otherwise call `Styler.render` to get the generated HTML.

CSS classes are attached to the generated HTML

- Index and Column names include `index_name` and `level<k>` where  $k$  is its level in a MultiIndex
- Index label cells include
  - `row_heading`
  - `row<n>` where  $n$  is the numeric position of the row
  - `level<k>` where  $k$  is the level in a MultiIndex
- Column label cells include `* col_heading * col<n>` where  $n$  is the numeric position of the column  
`* level<k>` where  $k$  is the level in a MultiIndex
- Blank cells include `blank`
- Data cells include `data`

## Attributes

<b>env</b>	(Jinja2 Environment)
<b>template</b>	(Jinja2 Template)
<b>loader</b>	(Jinja2 Loader)

## Methods

<code>apply(func[, axis, subset])</code>	Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.
<code>applymap(func[, subset])</code>	Apply a function elementwise, updating the HTML representation with the result.
<code>background_gradient([cmap, low, high, axis, ...])</code>	Color the background in a gradient according to the data in each column (optionally row).
<code>bar([subset, axis, color, width, align])</code>	Color the background <code>color</code> proportional to the values in each column.
<code>clear()</code>	“Reset” the styler, removing any previously applied styles.
<code>export()</code>	Export the styles to applied to the current Styler.
<code>format(formatter[, subset])</code>	Format the text display value of cells.
<code>from_custom_template(searchpath, name)</code>	Factory function for creating a subclass of <code>Styler</code> with a custom template and Jinja environment.
<code>hide_columns(subset)</code>	Hide columns from rendering.
<code>hide_index()</code>	Hide any indices from rendering.
<code>highlight_max([subset, color, axis])</code>	Highlight the maximum by shading the background
<code>highlight_min([subset, color, axis])</code>	Highlight the minimum by shading the background
<code>highlight_null([null_color])</code>	Shade the background <code>null_color</code> for missing values.
<code>render(**kwargs)</code>	Render the built up styles to HTML
<code>set_caption(caption)</code>	Set the caption on a Styler

Continued on next page

Table 175 – continued from previous page

<code>set_precision(precision)</code>	Set the precision used to render.
<code>set_properties([subset])</code>	Convenience method for setting one or more non-data dependent properties or each cell.
<code>set_table_attributes(attributes)</code>	Set the table attributes.
<code>set_table_styles(table_styles)</code>	Set the table styles on a Styler.
<code>set_uuid(uuid)</code>	Set the uuid for a Styler.
<code>to_excel(excel_writer[, sheet_name, na_rep, ...])</code>	Write Styler to an excel sheet
<code>use(styles)</code>	Set the styles on the current Styler, possibly using styles from <code>Styler.export</code> .
<code>where(cond, value[, other, subset])</code>	Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.

### **pandas.io.formats.style.Styler.apply**

`Styler.apply(func, axis=0, subset=None, **kwargs)`

Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.

**Parameters** `func` : function

`func` should take a Series or DataFrame (depending on `axis`), and return an object with the same shape. Must return a DataFrame with identical index and column labels when `axis=None`

**axis** : int, str or None

apply to each column (`axis=0` or `'index'`) or to each row (`axis=1` or `'columns'`) or to the entire DataFrame at once with `axis=None`

**subset** : IndexSlice

a valid indexer to limit data to *before* applying the function. Consider using a `pandas.IndexSlice`

**kwargs** : dict

pass along to `func`

**Returns**

`self` [Styler]

### **Notes**

The output shape of `func` should match the input, i.e. if `x` is the input row, column, or table (depending on `axis`), then `func(x.shape) == x.shape` should be true.

This is similar to `DataFrame.apply`, except that `axis=None` applies the function to the entire DataFrame at once, rather than column-wise or row-wise.

## Examples

```
>>> def highlight_max(x):
...     return ['background-color: yellow' if v == x.max() else ''
...            for v in x]
...
>>> df = pd.DataFrame(np.random.randn(5, 2))
>>> df.style.apply(highlight_max)
```

## pandas.io.formats.style.Styler.applymap

**Styler.applymap** (*func*, *subset=None*, *\*\*kwargs*)

Apply a function elementwise, updating the HTML representation with the result.

**Parameters** **func** : function

*func* should take a scalar and return a scalar

**subset** : IndexSlice

a valid indexer to limit data to *before* applying the function. Consider using a `pandas.IndexSlice`

**kwargs** : dict

pass along to *func*

**Returns**

**self** [Styler]

**See also:**

`Styler.where`

## pandas.io.formats.style.Styler.background\_gradient

**Styler.background\_gradient** (*cmap='PuBu'*, *low=0*, *high=0*, *axis=0*, *subset=None*)

Color the background in a gradient according to the data in each column (optionally row). Requires matplotlib.

**Parameters** **cmap**: str or colormap

matplotlib colormap

**low, high**: float

compress the range by these values.

**axis**: int or str

1 or 'columns' for columnwise, 0 or 'index' for rowwise

**subset**: IndexSlice

a valid slice for data to limit the style application to

**Returns**

**self** [Styler]

## Notes

Tune `low` and `high` to keep the text legible by not using the entire range of the color map. These extend the range of the data by `low * (x.max() - x.min())` and `high * (x.max() - x.min())` before normalizing.

## pandas.io.formats.style.Styler.bar

`Styler.bar(subset=None, axis=0, color='#d65f5f', width=100, align='left')`

Color the background `color` proportional to the values in each column. Excludes non-numeric data by default.

**Parameters** `subset`: `IndexSlice`, default `None`

a valid slice for `data` to limit the style application to

**axis**: `int`

**color**: `str` or `2-tuple/list`

If a `str` is passed, the color is the same for both negative and positive numbers. If `2-tuple/list` is used, the first element is the `color_negative` and the second is the `color_positive` (eg: `['#d65f5f', '#5fba7d']`)

**width**: `float`

A number between 0 or 100. The largest value will cover `width` percent of the cell's width

**align** : { 'left', 'zero', 'mid' }, default 'left'

- 'left' : the min value starts at the left of the cell
- 'zero' : a value of zero is located at the center of the cell
- 'mid' : the center of the cell is at  $(\text{max}-\text{min})/2$ , or if values are all negative (positive) the zero is aligned at the right (left) of the cell

New in version 0.20.0.

**Returns**

`self` [`Styler`]

## pandas.io.formats.style.Styler.clear

`Styler.clear()`

“Reset” the styler, removing any previously applied styles. Returns `None`.

## pandas.io.formats.style.Styler.export

`Styler.export()`

Export the styles to applied to the current `Styler`. Can be applied to a second style with `Styler.use`.

**Returns**

`styles`: `list`

See also:

`Styler.use`

### `pandas.io.formats.style.Styler.format`

`Styler.format` (*formatter*, *subset=None*)

Format the text display value of cells.

New in version 0.18.0.

#### Parameters

**formatter:** str, callable, or dict

**subset:** `IndexSlice`

An argument to `DataFrame.loc` that restricts which elements `formatter` is applied to.

#### Returns

**self** [`Styler`]

### Notes

`formatter` is either an `a` or a dict `{column name: a}` where `a` is one of

- str: this will be wrapped in: `a.format(x)`
- callable: called with the value of an individual cell

The default display value for numeric values is the “general” (g) format with `pd.options.display.precision`.

### Examples

```
>>> df = pd.DataFrame(np.random.randn(4, 2), columns=['a', 'b'])
>>> df.style.format("{:.2%}")
>>> df['c'] = ['a', 'b', 'c', 'd']
>>> df.style.format({'c': str.upper})
```

### `pandas.io.formats.style.Styler.from_custom_template`

**classmethod** `Styler.from_custom_template` (*searchpath*, *name*)

Factory function for creating a subclass of `Styler` with a custom template and Jinja environment.

**Parameters** **searchpath** : str or list

Path or paths of directories containing the templates

**name** : str

Name of your custom template to use for rendering

**Returns** **MyStyler** : subclass of `Styler`

has the correct `env` and `template` class attributes set.

### **pandas.io.formats.style.Styler.hide\_columns**

`Styler.hide_columns(subset)`

Hide columns from rendering.

New in version 0.23.0.

**Parameters** `subset: IndexSlice`

An argument to `DataFrame.loc` that identifies which columns are hidden.

**Returns**

`self` [Styler]

### **pandas.io.formats.style.Styler.hide\_index**

`Styler.hide_index()`

Hide any indices from rendering.

New in version 0.23.0.

**Returns**

`self` [Styler]

### **pandas.io.formats.style.Styler.highlight\_max**

`Styler.highlight_max(subset=None, color='yellow', axis=0)`

Highlight the maximum by shading the background

**Parameters** `subset: IndexSlice, default None`

a valid slice for `data` to limit the style application to

**color: str, default 'yellow'**

**axis: int, str, or None; default 0**

0 or 'index' for columnwise (default), 1 or 'columns' for rowwise, or None for tablewise

**Returns**

`self` [Styler]

### **pandas.io.formats.style.Styler.highlight\_min**

`Styler.highlight_min(subset=None, color='yellow', axis=0)`

Highlight the minimum by shading the background

**Parameters** `subset: IndexSlice, default None`

a valid slice for `data` to limit the style application to

**color: str, default 'yellow'**

**axis: int, str, or None; default 0**



0 or 'index' for columnwise (default), 1 or 'columns' for rowwise, or None for tablewise

#### Returns

**self** [Styler]

### pandas.io.formats.style.Styler.highlight\_null

**Styler.highlight\_null** (*null\_color='red'*)

Shade the background *null\_color* for missing values.

#### Parameters

**null\_color:** str

#### Returns

**self** [Styler]

### pandas.io.formats.style.Styler.render

**Styler.render** (*\*\*kwargs*)

Render the built up styles to HTML

#### Parameters **\*\*kwargs**:

Any additional keyword arguments are passed through to *self.template.render*. This is useful when you need to provide additional variables for a custom template.

New in version 0.20.

#### Returns **rendered:** str

the rendered HTML

### Notes

Styler objects have defined the *\_repr\_html\_* method which automatically calls *self.render()* when it's the last item in a Notebook cell. When calling *Styler.render()* directly, wrap the result in *IPython.display.HTML* to view the rendered HTML in the notebook.

Pandas uses the following keys in render. Arguments passed in *\*\*kwargs* take precedence, so think carefully if you want to override them:

- head
- cellstyle
- body
- uuid
- precision
- table\_styles
- caption
- table\_attributes

### pandas.io.formats.style.Styler.set\_caption

`Styler.set_caption(caption)`

Set the caption on a Styler

#### Parameters

**caption:** str

#### Returns

**self** [Styler]

### pandas.io.formats.style.Styler.set\_precision

`Styler.set_precision(precision)`

Set the precision used to render.

#### Parameters

**precision:** int

#### Returns

**self** [Styler]

### pandas.io.formats.style.Styler.set\_properties

`Styler.set_properties(subset=None, **kwargs)`

Convenience method for setting one or more non-data dependent properties on each cell.

#### Parameters

**subset:** IndexSlice

a valid slice for `data` to limit the style application to

**kwargs:** dict

property: value pairs to be set for each cell

#### Returns

**self** [Styler]

### Examples

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_properties(color="white", align="right")
>>> df.style.set_properties(**{'background-color': 'yellow'})
```

### pandas.io.formats.style.Styler.set\_table\_attributes

`Styler.set_table_attributes(attributes)`

Set the table attributes. These are the items that show up in the opening `<table>` tag in addition to the automatic (by default) id.

#### Parameters

**attributes** [string]

**Returns**

**self** [Styler]

**Examples**

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_table_attributes('class="pure-table"')
# ... <table class="pure-table"> ...
```

**pandas.io.formats.style.Styler.set\_table\_styles**

**Styler.set\_table\_styles** (*table\_styles*)

Set the table styles on a Styler. These are placed in a <style> tag before the generated HTML table.

**Parameters** *table\_styles*: list

Each individual *table\_style* should be a dictionary with *selector* and *props* keys. *selector* should be a CSS selector that the style will be applied to (automatically prefixed by the table's UUID) and *props* should be a list of tuples with (*attribute*, *value*).

**Returns**

**self** [Styler]

**Examples**

```
>>> df = pd.DataFrame(np.random.randn(10, 4))
>>> df.style.set_table_styles(
...     [{'selector': 'tr:hover',
...        'props': [('background-color', 'yellow')]}]
... )
```

**pandas.io.formats.style.Styler.set\_uuid**

**Styler.set\_uuid** (*uuid*)

Set the uuid for a Styler.

**Parameters**

**uuid**: str

**Returns**

**self** [Styler]

**pandas.io.formats.style.Styler.to\_excel**

```
Styler.to_excel(excel_writer, sheet_name='Sheet1', na_rep="", float_format=None,
                columns=None, header=True, index=True, index_label=None, startrow=0,
                startcol=0, engine=None, merge_cells=True, encoding=None, inf_rep='inf',
                verbose=True, freeze_panes=None)
```

Write Styler to an excel sheet

New in version 0.20.

**Parameters** **excel\_writer** : string or ExcelWriter object

File path or existing ExcelWriter

**sheet\_name** : string, default 'Sheet1'

Name of sheet which will contain DataFrame

**na\_rep** : string, default ''

Missing data representation

**float\_format** : string, default None

Format string for floating point numbers

**columns** : sequence, optional

Columns to write

**header** : boolean or list of string, default True

Write out the column names. If a list of strings is given it is assumed to be aliases for the column names

**index** : boolean, default True

Write row names (index)

**index\_label** : string or sequence, default None

Column label for index column(s) if desired. If None is given, and *header* and *index* are True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex.

**startrow** :

upper left cell row to dump data frame

**startcol** :

upper left cell column to dump data frame

**engine** : string, default None

write engine to use - you can also set this via the options `io.excel.xlsx.writer`, `io.excel.xls.writer`, and `io.excel.xlsm.writer`.

**merge\_cells** : boolean, default True

Write MultiIndex and Hierarchical Rows as merged cells.

**encoding**: string, default None

encoding of the resulting excel file. Only necessary for xlwt, other writers support unicode natively.

**inf\_rep** : string, default 'inf'

Representation for infinity (there is no native representation for infinity in Excel)

**freeze\_panes** : tuple of integer (length 2), default None

Specifies the one-based bottommost row and rightmost column that is to be frozen

New in version 0.20.0.

## Notes

If passing an existing ExcelWriter object, then the sheet will be added to the existing workbook. This can be used to save different DataFrames to one workbook:

```
>>> writer = pd.ExcelWriter('output.xlsx')
>>> df1.to_excel(writer, 'Sheet1')
>>> df2.to_excel(writer, 'Sheet2')
>>> writer.save()
```

For compatibility with `to_csv`, `to_excel` serializes lists and dicts to strings before writing.

## pandas.io.formats.style.Styler.use

`Styler.use` (*styles*)

Set the styles on the current Styler, possibly using styles from `Styler.export`.

**Parameters** **styles**: list

list of style functions

**Returns**

**self** [Styler]

**See also:**

`Styler.export`

## pandas.io.formats.style.Styler.where

`Styler.where` (*cond*, *value*, *other=None*, *subset=None*, *\*\*kwargs*)

Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.

New in version 0.21.0.

**Parameters** **cond** : callable

`cond` should take a scalar and return a boolean

**value** : str

applied when `cond` returns true

**other** : str

applied when `cond` returns false

**subset** : IndexSlice

a valid indexer to limit data to *before* applying the function. Consider using a `pandas.IndexSlice`

**kwargs** : dict

pass along to cond

**Returns****self** [Styler]**See also:***Styler.applymap*

## 34.19.2 Styler Attributes

*Styler.env**Styler.template**Styler.loader*

### 34.19.2.1 pandas.io.formats.style.Styler.env

`Styler.env = <jinja2.environment.Environment object>`

### 34.19.2.2 pandas.io.formats.style.Styler.template

`Styler.template = <Template 'html.tpl'>`

### 34.19.2.3 pandas.io.formats.style.Styler.loader

`Styler.loader = <jinja2.loaders.PackageLoader object>`

## 34.19.3 Style Application

<i>Styler.apply</i> (func[, axis, subset])	Apply a function column-wise, row-wise, or table-wise, updating the HTML representation with the result.
<i>Styler.applymap</i> (func[, subset])	Apply a function elementwise, updating the HTML representation with the result.
<i>Styler.where</i> (cond, value[, other, subset])	Apply a function elementwise, updating the HTML representation with a style which is selected in accordance with the return value of a function.
<i>Styler.format</i> (formatter[, subset])	Format the text display value of cells.
<i>Styler.set_precision</i> (precision)	Set the precision used to render.
<i>Styler.set_table_styles</i> (table_styles)	Set the table styles on a Styler.
<i>Styler.set_table_attributes</i> (attributes)	Set the table attributes.
<i>Styler.set_caption</i> (caption)	Set the caption on a Styler
<i>Styler.set_properties</i> ([subset])	Convenience method for setting one or more non-data dependent properties or each cell.
<i>Styler.set_uuid</i> (uuid)	Set the uuid for a Styler.
<i>Styler.clear</i> ()	“Reset” the styler, removing any previously applied styles.

### 34.19.4 Builtin Styles

<code>Styler.highlight_max([subset, color, axis])</code>	Highlight the maximum by shading the background
<code>Styler.highlight_min([subset, color, axis])</code>	Highlight the minimum by shading the background
<code>Styler.highlight_null([null_color])</code>	Shade the background <code>null_color</code> for missing values.
<code>Styler.background_gradient([cmap, low, ...])</code>	Color the background in a gradient according to the data in each column (optionally row).
<code>Styler.bar([subset, axis, color, width, align])</code>	Color the background <code>color</code> proportional to the values in each column.

### 34.19.5 Style Export and Import

<code>Styler.render(**kwargs)</code>	Render the built up styles to HTML
<code>Styler.export()</code>	Export the styles to applied to the current Styler.
<code>Styler.use(styles)</code>	Set the styles on the current Styler, possibly using styles from <code>Styler.export</code> .
<code>Styler.to_excel(excel_writer[, sheet_name, ...])</code>	Write Styler to an excel sheet

## 34.20 Plotting

The following functions are contained in the `pandas.plotting` module.

<code>andrews_curves(frame, class_column[, ax, ...])</code>	Generates a matplotlib plot of Andrews curves, for visualising clusters of multivariate data.
<code>bootstrap_plot(series[, fig, size, samples])</code>	Bootstrap plot on mean, median and mid-range statistics.
<code>deregister_matplotlib_converters()</code>	Remove pandas' formatters and converters
<code>lag_plot(series[, lag, ax])</code>	Lag plot for time series.
<code>parallel_coordinates(frame, class_column[, ...])</code>	Parallel coordinates plotting.
<code>radviz(frame, class_column[, ax, color, ...])</code>	Plot a multidimensional dataset in 2D.
<code>register_matplotlib_converters([explicit])</code>	Register Pandas Formatters and Converters with matplotlib
<code>scatter_matrix(frame[, alpha, figsize, ax, ...])</code>	Draw a matrix of scatter plots.

### 34.20.1 pandas.plotting.andrews\_curves

`pandas.plotting.andrews_curves` (*frame, class\_column, ax=None, samples=200, color=None, colormap=None, \*\*kws*)

Generates a matplotlib plot of Andrews curves, for visualising clusters of multivariate data.

Andrews curves have the functional form:

$$f(t) = x_1/\sqrt{2} + x_2 \sin(t) + x_3 \cos(t) + x_4 \sin(2t) + x_5 \cos(2t) + \dots$$

Where  $x$  coefficients correspond to the values of each dimension and  $t$  is linearly spaced between  $-\pi$  and  $+\pi$ . Each row of `frame` then corresponds to a single curve.

**Parameters** `frame` : DataFrame

Data to be plotted, preferably normalized to (0.0, 1.0)

**class\_column** [Name of the column containing class names]

**ax** [matplotlib axes object, default None]

**samples** [Number of points to plot in each curve]

**color: list or tuple, optional**

Colors to use for the different classes

**colormap** : str or matplotlib colormap object, default None

Colormap to select colors from. If string, load colormap with that name from matplotlib.

**kwds: keywords**

Options to pass to matplotlib plotting method

**Returns**

**ax: Matplotlib axis object**

## 34.20.2 pandas.plotting.bootstrap\_plot

`pandas.plotting.bootstrap_plot` (*series*, *fig=None*, *size=50*, *samples=500*, *\*\*kwds*)

Bootstrap plot on mean, median and mid-range statistics.

The bootstrap plot is used to estimate the uncertainty of a statistic by relaying on random sampling with replacement [R33]. This function will generate bootstrapping plots for mean, median and mid-range statistics for the given number of samples of the given size.

**Parameters** **series** : pandas.Series

Pandas Series from where to get the samplings for the bootstrapping.

**fig** : matplotlib.figure.Figure, default None

If given, it will use the *fig* reference for plotting instead of creating a new one with default parameters.

**size** : int, default 50

Number of data points to consider during each sampling. It must be greater or equal than the length of the *series*.

**samples** : int, default 500

Number of times the bootstrap procedure is performed.

**\*\*kwds** :

Options to pass to matplotlib plotting method.

**Returns** **fig** : matplotlib.figure.Figure

Matplotlib figure

**See also:**

`pandas.DataFrame.plot` Basic plotting for DataFrame objects.

`pandas.Series.plot` Basic plotting for Series objects.



## Examples

```
>>> import numpy as np
>>> s = pd.Series(np.random.uniform(size=100))
>>> fig = pd.plotting.bootstrap_plot(s)
```

### 34.20.3 pandas.plotting.deregister\_matplotlib\_converters

`pandas.plotting.deregister_matplotlib_converters()`

Remove pandas' formatters and converters

Removes the custom converters added by `register()`. This attempts to set the state of the registry back to the state before pandas registered its own units. Converters for pandas' own types like `Timestamp` and `Period` are removed completely. Converters for types pandas overwrites, like `datetime.datetime`, are restored to their original value.

**See also:**

`deregister_matplotlib_converters`

### 34.20.4 pandas.plotting.lag\_plot

`pandas.plotting.lag_plot(series, lag=1, ax=None, **kws)`

Lag plot for time series.

#### Parameters

**series:** Time series

**lag:** lag of the scatter plot, default 1

**ax:** Matplotlib axis object, optional

**kws:** Matplotlib scatter method keyword arguments, optional

#### Returns

**ax:** Matplotlib axis object

### 34.20.5 pandas.plotting.parallel\_coordinates

`pandas.plotting.parallel_coordinates(frame, class_column, cols=None, ax=None, color=None, use_columns=False, xticks=None, colormap=None, axvlines=True, axvlines_kws=None, sort_labels=False, **kws)`

Parallel coordinates plotting.

#### Parameters

**frame:** DataFrame

**class\_column:** str

Column name containing class names

**cols:** list, optional

A list of column names to use

**ax:** matplotlib.axis, optional

matplotlib axis object

**color:** list or tuple, optional

Colors to use for the different classes

**use\_columns:** bool, optional

If true, columns will be used as xticks

**xticks:** list or tuple, optional

A list of values to use for xticks

**colormap:** str or matplotlib colormap, default None

Colormap to use for line colors.

**axvlines:** bool, optional

If true, vertical lines will be added at each xtick

**axvlines\_kwds:** keywords, optional

Options to be passed to axvline method for vertical lines

**sort\_labels:** bool, False

Sort class\_column labels, useful when assigning colors

New in version 0.20.0.

**kwds:** keywords

Options to pass to matplotlib plotting method

#### Returns

**ax:** matplotlib axis object

### Examples

```
>>> from pandas import read_csv
>>> from pandas.tools.plotting import parallel_coordinates
>>> from matplotlib import pyplot as plt
>>> df = read_csv('https://raw.githubusercontent.com/pandas-dev/pandas/master/
                '/pandas/tests/data/iris.csv')
>>> parallel_coordinates(df, 'Name', color=('#556270',
                '#4ECDC4', '#C7F464'))
>>> plt.show()
```

## 34.20.6 pandas.plotting.radviz

pandas.plotting.**radviz**(frame, class\_column, ax=None, color=None, colormap=None, \*\*kwds)

Plot a multidimensional dataset in 2D.

Each Series in the DataFrame is represented as a evenly distributed slice on a circle. Each data point is rendered in the circle according to the value on each Series. Highly correlated *Series* in the *DataFrame* are placed closer on the unit circle.

RadViz allow to project a N-dimensional data set into a 2D space where the influence of each dimension can be interpreted as a balance between the influence of all dimensions.

More info available at the [original article](#) describing RadViz.

**Parameters** `frame` : *DataFrame*

Pandas object holding the data.

**class\_column** : str

Column name containing the name of the data point category.

**ax** : `matplotlib.axes.Axes`, optional

A plot instance to which to add the information.

**color** : list[str] or tuple[str], optional

Assign a color to each category. Example: ['blue', 'green'].

**colormap** : str or `matplotlib.colors.Colormap`, default None

Colormap to select colors from. If string, load colormap with that name from matplotlib.

**kwds** : optional

Options to pass to matplotlib scatter plotting method.

**Returns**

**axes** [`matplotlib.axes.Axes`]

**See also:**

[`pandas.plotting.andrews\_curves`](#) Plot clustering visualization

## Examples

```
>>> df = pd.DataFrame({
...     'SepalLength': [6.5, 7.7, 5.1, 5.8, 7.6, 5.0, 5.4, 4.6,
...                     6.7, 4.6],
...     'SepalWidth': [3.0, 3.8, 3.8, 2.7, 3.0, 2.3, 3.0, 3.2,
...                     3.3, 3.6],
...     'PetalLength': [5.5, 6.7, 1.9, 5.1, 6.6, 3.3, 4.5, 1.4,
...                     5.7, 1.0],
...     'PetalWidth': [1.8, 2.2, 0.4, 1.9, 2.1, 1.0, 1.5, 0.2,
...                     2.1, 0.2],
...     'Category': ['virginica', 'virginica', 'setosa',
...                  'virginica', 'virginica', 'versicolor',
...                  'versicolor', 'setosa', 'virginica',
...                  'setosa']
... })
>>> rad_viz = pd.plotting.radviz(df, 'Category')
```

### 34.20.7 pandas.plotting.register\_matplotlib\_converters

`pandas.plotting.register_matplotlib_converters` (*explicit=True*)

Register Pandas Formatters and Converters with matplotlib

This function modifies the global `matplotlib.units.registry` dictionary. Pandas adds custom converters for

- `pd.Timestamp`
- `pd.Period`
- `np.datetime64`
- `datetime.datetime`
- `datetime.date`
- `datetime.time`

**See also:**

`deregister_matplotlib_converter`

### 34.20.8 `pandas.plotting.scatter_matrix`

`pandas.plotting.scatter_matrix` (*frame*, *alpha=0.5*, *figsize=None*, *ax=None*, *grid=False*, *diagonal='hist'*, *marker='.'*, *density\_kwds=None*, *hist\_kwds=None*, *range\_padding=0.05*, *\*\*kwds*)

Draw a matrix of scatter plots.

#### Parameters

**frame** [DataFrame]

**alpha** : float, optional

amount of transparency applied

**figsize** : (float,float), optional

a tuple (width, height) in inches

**ax** [Matplotlib axis object, optional]

**grid** : bool, optional

setting this to True will show the grid

**diagonal** : { 'hist', 'kde' }

pick between 'kde' and 'hist' for either Kernel Density Estimation or Histogram plot in the diagonal

**marker** : str, optional

Matplotlib marker type, default '.'

**hist\_kwds** : other plotting keyword arguments

To be passed to hist function

**density\_kwds** : other plotting keyword arguments

To be passed to kernel density estimate plot

**range\_padding** : float, optional

relative extension of axis range in x and y with respect to (x\_max - x\_min) or (y\_max - y\_min), default 0.05

**kws** : other plotting keyword arguments

To be passed to scatter function

### Examples

```
>>> df = DataFrame(np.random.randn(1000, 4), columns=['A', 'B', 'C', 'D'])
>>> scatter_matrix(df, alpha=0.2)
```

## 34.21 General utility functions

### 34.21.1 Working with options

<code>describe_option(pat[, _print_desc])</code>	Prints the description for one or more registered options.
<code>reset_option(pat)</code>	Reset one or more options to their default value.
<code>get_option(pat)</code>	Retrieves the value of the specified option.
<code>set_option(pat, value)</code>	Sets the value of the specified option.
<code>option_context(*args)</code>	Context manager to temporarily set options in the <i>with</i> statement context.

#### 34.21.1.1 pandas.describe\_option

`pandas.describe_option(pat, _print_desc=False)` = `<pandas.core.config.CallableDynamicDoc object>`

Prints the description for one or more registered options.

Call with not arguments to get a listing for all registered options.

Available options:

- `compute.[use_bottleneck, use_numexpr]`
- `display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]`
- `display.html.[border, table_schema, use_mathjax]`
- `display.[large_repr]`
- `display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]`
- `display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]`
- `display.unicode.[ambiguous_as_wide, east_asian_width]`
- `display.[width]`
- `html.[border]`
- `io.excel.xls.[writer]`
- `io.excel.xlsm.[writer]`
- `io.excel.xlsx.[writer]`

- `io.hdf.[default_format, dropna_table]`
- `io.parquet.[engine]`
- `mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]`
- `plotting.matplotlib.[register_converters]`

**Parameters** `pat` : str

Regexp pattern. All matching keys will have their description displayed.

`_print_desc` : bool, default True

If True (default) the description(s) will be printed to stdout. Otherwise, the description(s) will be returned as a unicode string (for testing).

**Returns**

None by default, the description(s) as a unicode string if `_print_desc` is False

## Notes

The available options with its descriptions:

**compute.use\_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True  
Valid values: False, True [default: True] [currently: True]

**compute.use\_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True  
Valid values: False, True [default: True] [currently: True]

**display.chop\_threshold** [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

**display.colheader\_justify** ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

**display.column\_space** No description available. [default: 12] [currently: 12]

**display.date\_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

**display.date\_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

**display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by `to_string`, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

**display.expand\_frame\_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, `max_columns` is still respected, but the output will wrap-around across multiple “pages” if its width exceeds `display.width`. [default: True] [currently: True]

**display.float\_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See `formats.format.EngFormatter` for an example. [default: None] [currently: None]

**display.html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

- display.html.table\_schema** [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]
- display.html.use\_mathjax** [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]
- display.large\_repr** ['truncate'/'info'] For DataFrames exceeding max\_rows/max\_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]
- display.latex.escape** [bool] This specifies if the to\_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]
- display.latex.longtable** :bool This specifies if the to\_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]
- display.latex.multicolumn** [bool] This specifies if the to\_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]
- display.latex.multicolumn\_format** [bool] This specifies if the to\_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]
- display.latex.mulirow** [bool] This specifies if the to\_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]
- display.latex.repr** [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]
- display.max\_categories** [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype "category". [default: 8] [currently: 8]
- display.max\_columns** [int] If max\_cols is exceeded, switch to truncate view. Depending on *large\_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.
- In case python/IPython is running in a terminal and *large\_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]
- display.max\_colwidth** [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a "..." placeholder is embedded in the output. [default: 50] [currently: 50]
- display.max\_info\_columns** [int] max\_info\_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]
- display.max\_info\_rows** [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max\_info\_rows and max\_info\_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]
- display.max\_rows** [int] If max\_rows is exceeded, switch to truncate view. Depending on *large\_repr*, objects are either centrally truncated or printed as a summary view. 'None' value means unlimited.
- In case python/IPython is running in a terminal and *large\_repr* equals 'truncate' this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]
- display.max\_seq\_items** [int or None] when pretty-printing a long sequence, no more than *max\_seq\_items* will be printed. If items are omitted, they will be denoted by the addition of "..." to the resulting string.
- If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

- display.memory\_usage** [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when `df.info()` is called. Valid values `True`, `False`, `'deep'` [default: `True`] [currently: `True`]
- display.multi\_sparse** [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: `True`] [currently: `True`]
- display.notebook\_repr\_html** [boolean] When `True`, IPython notebook will use html representation for pandas objects (if it is available). [default: `True`] [currently: `True`]
- display.pprint\_nest\_depth** [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]
- display.precision** [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]
- display.show\_dimensions** [boolean or `'truncate'`] Whether to print out dimensions at the end of DataFrame repr. If `'truncate'` is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: `truncate`] [currently: `truncate`]
- display.unicode.ambiguous\_as\_wide** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: `False`) [default: `False`] [currently: `False`]
- display.unicode.east\_asian\_width** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: `False`) [default: `False`] [currently: `False`]
- display.width** [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to `None` and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]
- html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use `display.html.border` instead.)
- io.excel.xls.writer** [string] The default Excel writer engine for `'xls'` files. Available options: `auto`, `xlwt`. [default: `auto`] [currently: `auto`]
- io.excel.xlsm.writer** [string] The default Excel writer engine for `'xlsm'` files. Available options: `auto`, `openpyxl`. [default: `auto`] [currently: `auto`]
- io.excel.xlsx.writer** [string] The default Excel writer engine for `'xlsx'` files. Available options: `auto`, `openpyxl`, `xlsxwriter`. [default: `auto`] [currently: `auto`]
- io.hdf.default\_format** [format] default format writing format, if `None`, then put will default to `'fixed'` and append will default to `'table'` [default: `None`] [currently: `None`]
- io.hdf.dropna\_table** [boolean] drop ALL nan rows when appending to a table [default: `False`] [currently: `False`]
- io.parquet.engine** [string] The default parquet reader/writer engine. Available options: `'auto'`, `'pyarrow'`, `'fastparquet'`, the default is `'auto'` [default: `auto`] [currently: `auto`]
- mode.chained\_assignment** [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: `warn`] [currently: `warn`]
- mode.sim\_interactive** [boolean] Whether to simulate interactive mode for purposes of testing [default: `False`] [currently: `False`]
- mode.use\_inf\_as\_na** [boolean] `True` means treat `None`, `NaN`, `INF`, `-INF` as `NA` (old way), `False` means `None` and `NaN` are null, but `INF`, `-INF` are not `NA` (new way). [default: `False`] [currently: `False`]



**mode.use\_inf\_as\_null** [boolean] `use_inf_as_null` had been deprecated and will be removed in a future version. Use `use_inf_as_na` instead. [default: False] [currently: False] (Deprecated, use `mode.use_inf_as_na` instead.)

**plotting.matplotlib.register\_converters** [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

### 34.21.1.2 pandas.reset\_option

`pandas.reset_option(pat) = <pandas.core.config.CallableDynamicDoc object>`

Reset one or more options to their default value.

Pass "all" as argument to reset all options.

Available options:

- `compute.[use_bottleneck, use_numexpr]`
- `display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]`
- `display.html.[border, table_schema, use_mathjax]`
- `display.[large_repr]`
- `display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]`
- `display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]`
- `display.unicode.[ambiguous_as_wide, east_asian_width]`
- `display.[width]`
- `html.[border]`
- `io.excel.xls.[writer]`
- `io.excel.xlsm.[writer]`
- `io.excel.xlsx.[writer]`
- `io.hdf.[default_format, dropna_table]`
- `io.parquet.[engine]`
- `mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]`
- `plotting.matplotlib.[register_converters]`

**Parameters** `pat` : str/regex

If specified only options matching *prefix\** will be reset. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. `x.y.z.option_name`), your code may break in future versions if new options with similar names are introduced.

**Returns**

None

## Notes

The available options with its descriptions:

**compute.use\_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True  
Valid values: False,True [default: True] [currently: True]

**compute.use\_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True  
Valid values: False,True [default: True] [currently: True]

**display.chop\_threshold** [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

**display.colheader\_justify** ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

**display.column\_space** **No description available.** [default: 12] [currently: 12]

**display.date\_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

**display.date\_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

**display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by to\_string, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

**display.expand\_frame\_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, *max\_columns* is still respected, but the output will wrap-around across multiple “pages” if its width exceeds *display.width*. [default: True] [currently: True]

**display.float\_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like SeriesFormatter. See formats.format.EngFormatter for an example. [default: None] [currently: None]

**display.html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

**display.html.table\_schema** [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]

**display.html.use\_mathjax** [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]

**display.large\_repr** ['truncate'/'info'] For DataFrames exceeding *max\_rows*/*max\_cols*, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from `df.info()` (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

**display.latex.escape** [bool] This specifies if the `to_latex` method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]

**display.latex.longtable** :bool This specifies if the `to_latex` method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]

**display.latex.multicolumn** [bool] This specifies if the `to_latex` method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]

**display.latex.multicolumn\_format** [bool] This specifies if the `to_latex` method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: 1] [currently: 1]

**display.latex.multirow** [bool] This specifies if the `to_latex` method of a DataFrame uses multirows to pretty-print MultiIndex rows. Valid values: False, True [default: False] [currently: False]

**display.latex.repr** [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]

**display.max\_categories** [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype “category”. [default: 8] [currently: 8]

**display.max\_columns** [int] If `max_cols` is exceeded, switch to truncate view. Depending on *large\_repr*, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and *large\_repr* equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

**display.max\_colwidth** [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a “...” placeholder is embedded in the output. [default: 50] [currently: 50]

**display.max\_info\_columns** [int] `max_info_columns` is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

**display.max\_info\_rows** [int or None] `df.info()` will usually show null-counts for each column. For large frames this can be quite slow. `max_info_rows` and `max_info_cols` limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

**display.max\_rows** [int] If `max_rows` is exceeded, switch to truncate view. Depending on *large\_repr*, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and *large\_repr* equals ‘truncate’ this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]

**display.max\_seq\_items** [int or None] when pretty-printing a long sequence, no more than *max\_seq\_items* will be printed. If items are omitted, they will be denoted by the addition of “...” to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

**display.memory\_usage** [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when `df.info()` is called. Valid values True, False, ‘deep’ [default: True] [currently: True]

**display.multi\_sparse** [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]

**display.notebook\_repr\_html** [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]

**display.pprint\_nest\_depth** [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]

**display.precision** [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]

**display.show\_dimensions** [boolean or ‘truncate’] Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]

**display.unicode.ambiguous\_as\_wide** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]

- display.unicode.east\_asian\_width** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]
- display.width** [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]
- html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use `display.html.border` instead.)
- io.excel.xls.writer** [string] The default Excel writer engine for 'xls' files. Available options: auto, xlwt. [default: auto] [currently: auto]
- io.excel.xlsm.writer** [string] The default Excel writer engine for 'xlsm' files. Available options: auto, openpyxl. [default: auto] [currently: auto]
- io.excel.xlsx.writer** [string] The default Excel writer engine for 'xlsx' files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]
- io.hdf.default\_format** [format] default format writing format, if None, then put will default to 'fixed' and append will default to 'table' [default: None] [currently: None]
- io.hdf.dropna\_table** [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]
- io.parquet.engine** [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]
- mode.chained\_assignment** [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]
- mode.sim\_interactive** [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]
- mode.use\_inf\_as\_na** [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]
- mode.use\_inf\_as\_null** [boolean] `use_inf_as_null` had been deprecated and will be removed in a future version. Use `use_inf_as_na` instead. [default: False] [currently: False] (Deprecated, use `mode.use_inf_as_na` instead.)
- plotting.matplotlib.register\_converters** [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

### 34.21.1.3 pandas.get\_option

`pandas.get_option(pat) = <pandas.core.config.CallableDynamicDoc object>`

Retrieves the value of the specified option.

Available options:

- `compute.[use_bottleneck, use_numexpr]`
- `display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]`
- `display.html.[border, table_schema, use_mathjax]`
- `display.[large_repr]`

- `display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]`
- `display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]`
- `display.unicode.[ambiguous_as_wide, east_asian_width]`
- `display.[width]`
- `html.[border]`
- `io.excel.xls.[writer]`
- `io.excel.xlsm.[writer]`
- `io.excel.xlsx.[writer]`
- `io.hdf.[default_format, dropna_table]`
- `io.parquet.[engine]`
- `mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]`
- `plotting.matplotlib.[register_converters]`

**Parameters** `pat` : str

Regex which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. `x.y.z.option_name`), your code may break in future versions if new options with similar names are introduced.

**Returns**

**result** [the value of the option]

**Raises**

**OptionError** [if no such option exists]

**Notes**

The available options with its descriptions:

**compute.use\_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True  
Valid values: False, True [default: True] [currently: True]

**compute.use\_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True  
Valid values: False, True [default: True] [currently: True]

**display.chop\_threshold** [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

**display.colheader\_justify** ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

**display.column\_space** No description available. [default: 12] [currently: 12]

**display.date\_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

**display.date\_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

- display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by `to_string`, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]
- display.expand\_frame\_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, `max_columns` is still respected, but the output will wrap-around across multiple “pages” if its width exceeds `display.width`. [default: True] [currently: True]
- display.float\_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like `SeriesFormatter`. See `formats.format.EngFormatter` for an example. [default: None] [currently: None]
- display.html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]
- display.html.table\_schema** [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]
- display.html.use\_mathjax** [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]
- display.large\_repr** ['truncate'/'info'] For DataFrames exceeding `max_rows/max_cols`, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from `df.info()` (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]
- display.latex.escape** [bool] This specifies if the `to_latex` method of a Dataframe uses escapes special characters. Valid values: False, True [default: True] [currently: True]
- display.latex.longtable** :bool This specifies if the `to_latex` method of a Dataframe uses the longtable format. Valid values: False, True [default: False] [currently: False]
- display.latex.multicolumn** [bool] This specifies if the `to_latex` method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False, True [default: True] [currently: True]
- display.latex.multicolumn\_format** [bool] This specifies if the `to_latex` method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False, True [default: 1] [currently: 1]
- display.latex.multirow** [bool] This specifies if the `to_latex` method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False, True [default: False] [currently: False]
- display.latex.repr** [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]
- display.max\_categories** [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype “category”. [default: 8] [currently: 8]
- display.max\_columns** [int] If `max_cols` is exceeded, switch to truncate view. Depending on `large_repr`, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.
- In case python/IPython is running in a terminal and `large_repr` equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]
- display.max\_colwidth** [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a “...” placeholder is embedded in the output. [default: 50] [currently: 50]
- display.max\_info\_columns** [int] `max_info_columns` is used in `DataFrame.info` method to decide if per column information will be printed. [default: 100] [currently: 100]

- display.max\_info\_rows** [int or None] `df.info()` will usually show null-counts for each column. For large frames this can be quite slow. `max_info_rows` and `max_info_cols` limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]
- display.max\_rows** [int] If `max_rows` is exceeded, switch to truncate view. Depending on `large_repr`, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.
- In case python/IPython is running in a terminal and `large_repr` equals ‘truncate’ this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]
- display.max\_seq\_items** [int or None] when pretty-printing a long sequence, no more than `max_seq_items` will be printed. If items are omitted, they will be denoted by the addition of “...” to the resulting string.
- If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]
- display.memory\_usage** [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when `df.info()` is called. Valid values True,False,’deep’ [default: True] [currently: True]
- display.multi\_sparse** [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]
- display.notebook\_repr\_html** [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]
- display.pprint\_nest\_depth** [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]
- display.precision** [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]
- display.show\_dimensions** [boolean or ‘truncate’] Whether to print out dimensions at the end of DataFrame repr. If ‘truncate’ is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]
- display.unicode.ambiguous\_as\_wide** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]
- display.unicode.east\_asian\_width** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]
- display.width** [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]
- html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use `display.html.border` instead.)
- io.excel.xls.writer** [string] The default Excel writer engine for ‘xls’ files. Available options: auto, xlwt. [default: auto] [currently: auto]
- io.excel.xlsm.writer** [string] The default Excel writer engine for ‘xlsm’ files. Available options: auto, openpyxl. [default: auto] [currently: auto]
- io.excel.xlsx.writer** [string] The default Excel writer engine for ‘xlsx’ files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]
- io.hdf.default\_format** [format] default format writing format, if None, then put will default to ‘fixed’ and append will default to ‘table’ [default: None] [currently: None]

- io.hdf.dropna\_table** [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]
- io.parquet.engine** [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]
- mode.chained\_assignment** [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]
- mode.sim\_interactive** [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]
- mode.use\_inf\_as\_na** [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]
- mode.use\_inf\_as\_null** [boolean] `use_inf_as_null` had been deprecated and will be removed in a future version. Use `use_inf_as_na` instead. [default: False] [currently: False] (Deprecated, use `mode.use_inf_as_na` instead.)
- plotting.matplotlib.register\_converters** [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

#### 34.21.1.4 pandas.set\_option

```
pandas.set_option(pat, value) = <pandas.core.config.CallableDynamicDoc
                        object>
```

Sets the value of the specified option.

Available options:

- `compute.[use_bottleneck, use_numexpr]`
- `display.[chop_threshold, colheader_justify, column_space, date_dayfirst, date_yearfirst, encoding, expand_frame_repr, float_format]`
- `display.html.[border, table_schema, use_mathjax]`
- `display.[large_repr]`
- `display.latex.[escape, longtable, multicolumn, multicolumn_format, multirow, repr]`
- `display.[max_categories, max_columns, max_colwidth, max_info_columns, max_info_rows, max_rows, max_seq_items, memory_usage, multi_sparse, notebook_repr_html, pprint_nest_depth, precision, show_dimensions]`
- `display.unicode.[ambiguous_as_wide, east_asian_width]`
- `display.[width]`
- `html.[border]`
- `io.excel.xls.[writer]`
- `io.excel.xlsm.[writer]`
- `io.excel.xlsx.[writer]`
- `io.hdf.[default_format, dropna_table]`
- `io.parquet.[engine]`
- `mode.[chained_assignment, sim_interactive, use_inf_as_na, use_inf_as_null]`
- `plotting.matplotlib.[register_converters]`



**Parameters** `pat` : str

Regex which should match a single option. Note: partial matches are supported for convenience, but unless you use the full option name (e.g. `x.y.z.option_name`), your code may break in future versions if new options with similar names are introduced.

**value** :

new value of option.

**Returns**

None

**Raises**

**OptionError** if no such option exists

**Notes**

The available options with its descriptions:

**compute.use\_bottleneck** [bool] Use the bottleneck library to accelerate if it is installed, the default is True  
Valid values: False,True [default: True] [currently: True]

**compute.use\_numexpr** [bool] Use the numexpr library to accelerate computation if it is installed, the default is True  
Valid values: False,True [default: True] [currently: True]

**display.chop\_threshold** [float or None] if set to a float value, all float values smaller then the given threshold will be displayed as exactly 0 by repr and friends. [default: None] [currently: None]

**display.colheader\_justify** ['left'/'right'] Controls the justification of column headers. used by DataFrameFormatter. [default: right] [currently: right]

**display.column\_space** No description available. [default: 12] [currently: 12]

**display.date\_dayfirst** [boolean] When True, prints and parses dates with the day first, eg 20/01/2005 [default: False] [currently: False]

**display.date\_yearfirst** [boolean] When True, prints and parses dates with the year first, eg 2005/01/20 [default: False] [currently: False]

**display.encoding** [str/unicode] Defaults to the detected encoding of the console. Specifies the encoding to be used for strings returned by `to_string`, these are generally strings meant to be displayed on the console. [default: UTF-8] [currently: UTF-8]

**display.expand\_frame\_repr** [boolean] Whether to print out the full DataFrame repr for wide DataFrames across multiple lines, `max_columns` is still respected, but the output will wrap-around across multiple “pages” if its width exceeds `display.width`. [default: True] [currently: True]

**display.float\_format** [callable] The callable should accept a floating point number and return a string with the desired format of the number. This is used in some places like `SeriesFormatter`. See `formats.format.EngFormatter` for an example. [default: None] [currently: None]

**display.html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1]

**display.html.table\_schema** [boolean] Whether to publish a Table Schema representation for frontends that support it. (default: False) [default: False] [currently: False]

**display.html.use\_mathjax** [boolean] When True, Jupyter notebook will process table contents using MathJax, rendering mathematical expressions enclosed by the dollar symbol. (default: True) [default: True] [currently: True]

**display.large\_repr** ['truncate'/'info'] For DataFrames exceeding max\_rows/max\_cols, the repr (and HTML repr) can show a truncated table (the default from 0.13), or switch to the view from df.info() (the behaviour in earlier versions of pandas). [default: truncate] [currently: truncate]

**display.latex.escape** [bool] This specifies if the to\_latex method of a Dataframe uses escapes special characters. Valid values: False,True [default: True] [currently: True]

**display.latex.longtable** :bool This specifies if the to\_latex method of a Dataframe uses the longtable format. Valid values: False,True [default: False] [currently: False]

**display.latex.multicolumn** [bool] This specifies if the to\_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: True] [currently: True]

**display.latex.multicolumn\_format** [bool] This specifies if the to\_latex method of a Dataframe uses multicolumns to pretty-print MultiIndex columns. Valid values: False,True [default: l] [currently: l]

**display.latex.mulirow** [bool] This specifies if the to\_latex method of a Dataframe uses multirows to pretty-print MultiIndex rows. Valid values: False,True [default: False] [currently: False]

**display.latex.repr** [boolean] Whether to produce a latex DataFrame representation for jupyter environments that support it. (default: False) [default: False] [currently: False]

**display.max\_categories** [int] This sets the maximum number of categories pandas should output when printing out a *Categorical* or a Series of dtype “category”. [default: 8] [currently: 8]

**display.max\_columns** [int] If max\_cols is exceeded, switch to truncate view. Depending on *large\_repr*, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and *large\_repr* equals ‘truncate’ this can be set to 0 and pandas will auto-detect the width of the terminal and print a truncated object which fits the screen width. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 0] [currently: 0]

**display.max\_colwidth** [int] The maximum width in characters of a column in the repr of a pandas data structure. When the column overflows, a “...” placeholder is embedded in the output. [default: 50] [currently: 50]

**display.max\_info\_columns** [int] max\_info\_columns is used in DataFrame.info method to decide if per column information will be printed. [default: 100] [currently: 100]

**display.max\_info\_rows** [int or None] df.info() will usually show null-counts for each column. For large frames this can be quite slow. max\_info\_rows and max\_info\_cols limit this null check only to frames with smaller dimensions than specified. [default: 1690785] [currently: 1690785]

**display.max\_rows** [int] If max\_rows is exceeded, switch to truncate view. Depending on *large\_repr*, objects are either centrally truncated or printed as a summary view. ‘None’ value means unlimited.

In case python/IPython is running in a terminal and *large\_repr* equals ‘truncate’ this can be set to 0 and pandas will auto-detect the height of the terminal and print a truncated object which fits the screen height. The IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to do correct auto-detection. [default: 60] [currently: 15]

**display.max\_seq\_items** [int or None] when pretty-printing a long sequence, no more than *max\_seq\_items* will be printed. If items are omitted, they will be denoted by the addition of “...” to the resulting string.

If set to None, the number of items to be printed is unlimited. [default: 100] [currently: 100]

**display.memory\_usage** [bool, string or None] This specifies if the memory usage of a DataFrame should be displayed when df.info() is called. Valid values True,False,’deep’ [default: True] [currently: True]

**display.multi\_sparse** [boolean] “sparsify” MultiIndex display (don’t display repeated elements in outer levels within groups) [default: True] [currently: True]

- display.notebook\_repr\_html** [boolean] When True, IPython notebook will use html representation for pandas objects (if it is available). [default: True] [currently: True]
- display.pprint\_nest\_depth** [int] Controls the number of nested levels to process when pretty-printing [default: 3] [currently: 3]
- display.precision** [int] Floating point output precision (number of significant digits). This is only a suggestion [default: 6] [currently: 6]
- display.show\_dimensions** [boolean or 'truncate'] Whether to print out dimensions at the end of DataFrame repr. If 'truncate' is specified, only print out the dimensions if the frame is truncated (e.g. not display all rows and/or columns) [default: truncate] [currently: truncate]
- display.unicode.ambiguous\_as\_wide** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]
- display.unicode.east\_asian\_width** [boolean] Whether to use the Unicode East Asian Width to calculate the display text width. Enabling this may affect to the performance (default: False) [default: False] [currently: False]
- display.width** [int] Width of the display in characters. In case python/IPython is running in a terminal this can be set to None and pandas will correctly auto-detect the width. Note that the IPython notebook, IPython qtconsole, or IDLE do not run in a terminal and hence it is not possible to correctly detect the width. [default: 80] [currently: 80]
- html.border** [int] A `border=value` attribute is inserted in the `<table>` tag for the DataFrame HTML repr. [default: 1] [currently: 1] (Deprecated, use `display.html.border` instead.)
- io.excel.xls.writer** [string] The default Excel writer engine for 'xls' files. Available options: auto, xlwt. [default: auto] [currently: auto]
- io.excel.xlsm.writer** [string] The default Excel writer engine for 'xlsm' files. Available options: auto, openpyxl. [default: auto] [currently: auto]
- io.excel.xlsx.writer** [string] The default Excel writer engine for 'xlsx' files. Available options: auto, openpyxl, xlsxwriter. [default: auto] [currently: auto]
- io.hdf.default\_format** [format] default format writing format, if None, then put will default to 'fixed' and append will default to 'table' [default: None] [currently: None]
- io.hdf.dropna\_table** [boolean] drop ALL nan rows when appending to a table [default: False] [currently: False]
- io.parquet.engine** [string] The default parquet reader/writer engine. Available options: 'auto', 'pyarrow', 'fastparquet', the default is 'auto' [default: auto] [currently: auto]
- mode.chained\_assignment** [string] Raise an exception, warn, or no action if trying to use chained assignment, The default is warn [default: warn] [currently: warn]
- mode.sim\_interactive** [boolean] Whether to simulate interactive mode for purposes of testing [default: False] [currently: False]
- mode.use\_inf\_as\_na** [boolean] True means treat None, NaN, INF, -INF as NA (old way), False means None and NaN are null, but INF, -INF are not NA (new way). [default: False] [currently: False]
- mode.use\_inf\_as\_null** [boolean] `use_inf_as_null` had been deprecated and will be removed in a future version. Use `use_inf_as_na` instead. [default: False] [currently: False] (Deprecated, use `mode.use_inf_as_na` instead.)
- plotting.matplotlib.register\_converters** [bool] Whether to register converters with matplotlib's units registry for dates, times, datetimes, and Periods. Toggling to False will remove the converters, restoring any converters that pandas overwrote. [default: True] [currently: True]

### 34.21.1.5 pandas.option\_context

**class** pandas.option\_context(\*args)

Context manager to temporarily set options in the *with* statement context.

You need to invoke as `option_context(pat, val, [(pat, val), ...])`.

#### Examples

```
>>> with option_context('display.max_rows', 10, 'display.max_columns', 5):  
    ...
```

### 34.21.2 Testing functions

<code>testing.assert_frame_equal(left,</code>	<code>right[,</code>	Check that left and right DataFrame are equal.
<code>...])</code>		
<code>testing.assert_series_equal(left,</code>	<code>right[,</code>	Check that left and right Series are equal.
<code>...])</code>		
<code>testing.assert_index_equal(left,</code>	<code>right[,</code>	Check that left and right Index are equal.
<code>...])</code>		

#### 34.21.2.1 pandas.testing.assert\_frame\_equal

`pandas.testing.assert_frame_equal(left, right, check_dtype=True, check_index_type='equiv',  
check_column_type='equiv', check_frame_type=True,  
check_less_precise=False, check_names=True,  
by_blocks=False, check_exact=False,  
check_datetimelike_compat=False,  
check_categorical=True, check_like=False,  
obj='DataFrame')`

Check that left and right DataFrame are equal.

##### Parameters

**left** [DataFrame]

**right** [DataFrame]

**check\_dtype** : bool, default True

Whether to check the DataFrame dtype is identical.

**check\_index\_type** : bool / string {'equiv'}, default False

Whether to check the Index class, dtype and inferred\_type are identical.

**check\_column\_type** : bool / string {'equiv'}, default False

Whether to check the columns class, dtype and inferred\_type are identical.

**check\_frame\_type** : bool, default False

Whether to check the DataFrame class is identical.

**check\_less\_precise** : bool or int, default False

Specify comparison precision. Only used when `check_exact` is `False`. 5 digits (`False`) or 3 digits (`True`) after decimal points are compared. If `int`, then specify the digits to compare

**check\_names** : bool, default `True`

Whether to check the Index names attribute.

**by\_blocks** : bool, default `False`

Specify how to compare internal data. If `False`, compare by columns. If `True`, compare by blocks.

**check\_exact** : bool, default `False`

Whether to compare number exactly.

**check\_datetimelike\_compat** : bool, default `False`

Compare datetime-like which is comparable ignoring dtype.

**check\_categorical** : bool, default `True`

Whether to compare internal Categorical exactly.

**check\_like** : bool, default `False`

If true, ignore the order of rows & columns

**obj** : str, default `'DataFrame'`

Specify object name being compared, internally used to show appropriate assertion message

### 34.21.2.2 pandas.testing.assert\_series\_equal

```
pandas.testing.assert_series_equal(left, right, check_dtype=True, check_index_type='equiv',
                                   check_series_type=True,      check_less_precise=False,
                                   check_names=True,             check_exact=False,
                                   check_datetimelike_compat=False,
                                   check_categorical=True, obj='Series')
```

Check that left and right Series are equal.

#### Parameters

**left** [Series]

**right** [Series]

**check\_dtype** : bool, default `True`

Whether to check the Series dtype is identical.

**check\_index\_type** : bool / string {'equiv'}, default `'equiv'`

Whether to check the Index class, dtype and inferred\_type are identical.

**check\_series\_type** : bool, default `True`

Whether to check the Series class is identical.

**check\_less\_precise** : bool or int, default `False`

Specify comparison precision. Only used when `check_exact` is `False`. 5 digits (`False`) or 3 digits (`True`) after decimal points are compared. If `int`, then specify the digits to compare

**check\_exact** : bool, default False

Whether to compare number exactly.

**check\_names** : bool, default True

Whether to check the Series and Index names attribute.

**check\_datetimelike\_compat** : bool, default False

Compare datetime-like which is comparable ignoring dtype.

**check\_categorical** : bool, default True

Whether to compare internal Categorical exactly.

**obj** : str, default 'Series'

Specify object name being compared, internally used to show appropriate assertion message

### 34.21.2.3 pandas.testing.assert\_index\_equal

```
pandas.testing.assert_index_equal(left, right, exact='equiv', check_names=True,  
                                  check_less_precise=False, check_exact=True,  
                                  check_categorical=True, obj='Index')
```

Check that left and right Index are equal.

#### Parameters

**left** [Index]

**right** [Index]

**exact** : bool / string { 'equiv' }, default False

Whether to check the Index class, dtype and inferred\_type are identical. If 'equiv', then RangeIndex can be substituted for Int64Index as well.

**check\_names** : bool, default True

Whether to check the names attribute.

**check\_less\_precise** : bool or int, default False

Specify comparison precision. Only used when check\_exact is False. 5 digits (False) or 3 digits (True) after decimal points are compared. If int, then specify the digits to compare

**check\_exact** : bool, default True

Whether to compare number exactly.

**check\_categorical** : bool, default True

Whether to compare internal Categorical exactly.

**obj** : str, default 'Index'

Specify object name being compared, internally used to show appropriate assertion message

### 34.21.3 Exceptions and warnings

<code>errors.DtypeWarning</code>	Warning raised when reading different dtypes in a column from a file.
<code>errors.EmptyDataError</code>	Exception that is thrown in <code>pd.read_csv</code> (by both the C and Python engines) when empty data or header is encountered.
<code>errors.OutOfBoundsDatetime</code>	
<code>errors.ParserError</code>	Exception that is raised by an error encountered in <code>pd.read_csv</code> .
<code>errors.ParserWarning</code>	Warning raised when reading a file that doesn't use the default 'c' parser.
<code>errors.PerformanceWarning</code>	Warning raised when there is a possible performance impact.
<code>errors.UnsortedIndexError</code>	Error raised when attempting to get a slice of a MultiIndex, and the index has not been lexsorted.
<code>errors.UnsupportedFunctionCall</code>	Exception raised when attempting to call a numpy function on a pandas object, but that function is not supported by the object e.g.

### 34.21.3.1 pandas.errors.DtypeWarning

#### exception `pandas.errors.DtypeWarning`

Warning raised when reading different dtypes in a column from a file.

Raised for a dtype incompatibility. This can happen whenever `read_csv` or `read_table` encounter non-uniform dtypes in a column(s) of a given CSV file.

See also:

`pandas.read_csv` Read CSV (comma-separated) file into a DataFrame.

`pandas.read_table` Read general delimited file into a DataFrame.

#### Notes

This warning is issued when dealing with larger files because the dtype checking happens per chunk read.

Despite the warning, the CSV file is read with mixed types in a single column which will be an object type. See the examples below to better understand this issue.

#### Examples

This example creates and reads a large CSV file with a column that contains *int* and *str*.

```
>>> df = pd.DataFrame({'a': ([ '1' ] * 100000 + [ 'X' ] * 100000 +
...                          [ '1' ] * 100000),
...                    'b': [ 'b' ] * 300000})
>>> df.to_csv('test.csv', index=False)
>>> df2 = pd.read_csv('test.csv')
... # DtypeWarning: Columns (0) have mixed types
```

Important to notice that `df2` will contain both *str* and *int* for the same input, '1'.

```
>>> df2.iloc[262140, 0]
'1'
>>> type(df2.iloc[262140, 0])
<class 'str'>
>>> df2.iloc[262150, 0]
1
>>> type(df2.iloc[262150, 0])
<class 'int'>
```

One way to solve this issue is using the *dtype* parameter in the *read\_csv* and *read\_table* functions to explicit the conversion:

```
>>> df2 = pd.read_csv('test.csv', sep=',', dtype={'a': str})
```

No warning was issued.

```
>>> import os
>>> os.remove('test.csv')
```

### 34.21.3.2 pandas.errors.EmptyDataError

**exception** pandas.errors.EmptyDataError

Exception that is thrown in *pd.read\_csv* (by both the C and Python engines) when empty data or header is encountered.

### 34.21.3.3 pandas.errors.OutOfBoundsDatetime

**exception** pandas.errors.OutOfBoundsDatetime

### 34.21.3.4 pandas.errors.ParserError

**exception** pandas.errors.ParserError

Exception that is raised by an error encountered in *pd.read\_csv*.

### 34.21.3.5 pandas.errors.ParserWarning

**exception** pandas.errors.ParserWarning

Warning raised when reading a file that doesn't use the default 'c' parser.

Raised by *pd.read\_csv* and *pd.read\_table* when it is necessary to change parsers, generally from the default 'c' parser to 'python'.

It happens due to a lack of support or functionality for parsing a particular attribute of a CSV file with the requested engine.

Currently, 'c' unsupported options include the following parameters:

1. *sep* other than a single character (e.g. regex separators)
2. *skipfooter* higher than 0
3. *sep=None* with *delim\_whitespace=False*



The warning can be avoided by adding `engine='python'` as a parameter in `pd.read_csv` and `pd.read_table` methods.

See also:

**pd.read\_csv** Read CSV (comma-separated) file into DataFrame.

**pd.read\_table** Read general delimited file into DataFrame.

## Examples

Using a `sep` in `pd.read_csv` other than a single character:

```
>>> import io
>>> csv = u'''a;b;c
...         1;1,8
...         1;2,1'''
>>> df = pd.read_csv(io.StringIO(csv), sep='[;,]')
... # ParserWarning: Falling back to the 'python' engine...
```

Adding `engine='python'` to `pd.read_csv` removes the Warning:

```
>>> df = pd.read_csv(io.StringIO(csv), sep='[;,]', engine='python')
```

### 34.21.3.6 pandas.errors.PerformanceWarning

**exception** `pandas.errors.PerformanceWarning`

Warning raised when there is a possible performance impact.

### 34.21.3.7 pandas.errors.UnsortedIndexError

**exception** `pandas.errors.UnsortedIndexError`

Error raised when attempting to get a slice of a MultiIndex, and the index has not been lexsorted. Subclass of `KeyError`.

New in version 0.20.0.

### 34.21.3.8 pandas.errors.UnsupportedFunctionCall

**exception** `pandas.errors.UnsupportedFunctionCall`

Exception raised when attempting to call a numpy function on a pandas object, but that function is not supported by the object e.g. `np.cumsum(groupby_object)`.

## 34.21.4 Data types related functionality

---

`api.types.union_categoricals(to_union[, ...])`

Combine list-like of Categorical-like, unioning categories.

---

`api.types.infer_dtype`

Efficiently infer the type of a passed val, or list-like array of values.

Continued on next page

Table 184 – continued from previous page

<code>pandas.api.types.pandas_dtype(dtype)</code>	Converts input into a pandas only dtype object or a numpy dtype object.
---	---

### 34.21.4.1 pandas.api.types.union\_categoricals

`pandas.api.types.union_categoricals` (*to\_union*, *sort\_categories=False*, *ignore\_order=False*)  
Combine list-like of Categorical-like, unioning categories. All categories must have the same dtype.

New in version 0.19.0.

**Parameters** *to\_union* : list-like of Categorical, CategoricalIndex,  
or Series with dtype='category'

**sort\_categories** : boolean, default False

If true, resulting categories will be lexicographically sorted, otherwise they will be ordered as they appear in the data.

**ignore\_order**: boolean, default False

If true, the ordered attribute of the Categoricals will be ignored. Results in an unordered categorical.

New in version 0.20.0.

#### Returns

**result** [Categorical]

#### Raises TypeError

- all inputs do not have the same dtype
- all inputs do not have the same ordered property
- all inputs are ordered and their categories are not identical
- *sort\_categories=True* and Categoricals are ordered

#### ValueError

Empty list of categoricals passed

#### Notes

To learn more about categories, see [link](#)

#### Examples

```
>>> from pandas.api.types import union_categoricals
```

If you want to combine categoricals that do not necessarily have the same categories, `union_categoricals` will combine a list-like of categoricals. The new categories will be the union of the categories being combined.

```
>>> a = pd.Categorical(["b", "c"])
>>> b = pd.Categorical(["a", "b"])
>>> union_categoricals([a, b])
```

(continues on next page)

(continued from previous page)

```
[b, c, a, b]
Categories (3, object): [b, c, a]
```

By default, the resulting categories will be ordered as they appear in the *categories* of the data. If you want the categories to be lexicographically sorted, use *sort\_categories=True* argument.

```
>>> union_categoricals([a, b], sort_categories=True)
[b, c, a, b]
Categories (3, object): [a, b, c]
```

*union\_categoricals* also works with the case of combining two categoricals of the same categories and order information (e.g. what you could also *append* for).

```
>>> a = pd.Categorical(["a", "b"], ordered=True)
>>> b = pd.Categorical(["a", "b", "a"], ordered=True)
>>> union_categoricals([a, b])
[a, b, a, b, a]
Categories (2, object): [a < b]
```

Raises *TypeError* because the categories are ordered and not identical.

```
>>> a = pd.Categorical(["a", "b"], ordered=True)
>>> b = pd.Categorical(["a", "b", "c"], ordered=True)
>>> union_categoricals([a, b])
TypeError: to union ordered Categoricals, all categories must be the same
```

New in version 0.20.0

Ordered categoricals with different categories or orderings can be combined by using the *ignore\_ordered=True* argument.

```
>>> a = pd.Categorical(["a", "b", "c"], ordered=True)
>>> b = pd.Categorical(["c", "b", "a"], ordered=True)
>>> union_categoricals([a, b], ignore_order=True)
[a, b, c, c, b, a]
Categories (3, object): [a, b, c]
```

*union\_categoricals* also works with a *CategoricalIndex*, or *Series* containing categorical data, but note that the resulting array will always be a plain *Categorical*

```
>>> a = pd.Series(["b", "c"], dtype='category')
>>> b = pd.Series(["a", "b"], dtype='category')
>>> union_categoricals([a, b])
[b, c, a, b]
Categories (3, object): [b, c, a]
```

#### 34.21.4.2 pandas.api.types.infer\_dtype

`pandas.api.types.infer_dtype()`

Efficiently infer the type of a passed val, or list-like array of values. Return a string describing the type.

##### Parameters

**value** [scalar, list, ndarray, or pandas type]

**skipna** : bool, default False

Ignore NaN values when inferring the type. The default of `False` will be deprecated in a later version of pandas.

New in version 0.21.0.

#### Returns

string describing the common type of the input data.

Results can include:

- string
- unicode
- bytes
- floating
- integer
- mixed-integer
- mixed-integer-float
- decimal
- complex
- categorical
- boolean
- datetime64
- datetime
- date
- timedelta64
- timedelta
- time
- period
- mixed

#### Raises

`TypeError` if ndarray-like but cannot infer the dtype

#### Notes

- ‘mixed’ is the catchall for anything that is not otherwise specialized
- ‘mixed-integer-float’ are floats and integers
- ‘mixed-integer’ are integers mixed with non-integers

## Examples

```
>>> infer_dtype(['foo', 'bar'])
'string'
```

```
>>> infer_dtype(['a', np.nan, 'b'], skipna=True)
'string'
```

```
>>> infer_dtype(['a', np.nan, 'b'], skipna=False)
'mixed'
```

```
>>> infer_dtype([b'foo', b'bar'])
'bytes'
```

```
>>> infer_dtype([1, 2, 3])
'integer'
```

```
>>> infer_dtype([1, 2, 3.5])
'mixed-integer-float'
```

```
>>> infer_dtype([1.0, 2.0, 3.5])
'floating'
```

```
>>> infer_dtype(['a', 1])
'mixed-integer'
```

```
>>> infer_dtype([Decimal(1), Decimal(2.0)])
'decimal'
```

```
>>> infer_dtype([True, False])
'boolean'
```

```
>>> infer_dtype([True, False, np.nan])
'mixed'
```

```
>>> infer_dtype([pd.Timestamp('20130101')])
'datetime'
```

```
>>> infer_dtype([datetime.date(2013, 1, 1)])
'date'
```

```
>>> infer_dtype([np.datetime64('2013-01-01')])
'datetime64'
```

```
>>> infer_dtype([datetime.timedelta(0, 1, 1)])
'timedelta'
```

```
>>> infer_dtype(pd.Series(list('aabc')).astype('category'))
'categorical'
```

### 34.21.4.3 pandas.api.types.pandas\_dtype

pandas.api.types.pandas\_dtype(dtype)

Converts input into a pandas only dtype object or a numpy dtype object.

#### Parameters

**dtype** [object to be converted]

#### Returns

**np.dtype or a pandas dtype**

Dtype introspection

<code>api.types.is_bool_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of a boolean dtype.
<code>api.types.is_categorical_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of the Categorical dtype.
<code>api.types.is_complex_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of a complex dtype.
<code>api.types.is_datetime64_any_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of the datetime64 dtype.
<code>api.types.is_datetime64_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of the datetime64 dtype.
<code>api.types.is_datetime64_ns_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of the datetime64[ns] dtype.
<code>api.types.is_datetime64tz_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of a DatetimeTZDtype dtype.
<code>api.types.is_extension_type(arr)</code>	Check whether an array-like is of a pandas extension class instance.
<code>api.types.is_float_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of a float dtype.
<code>api.types.is_int64_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of the int64 dtype.
<code>api.types.is_integer_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of an integer dtype.
<code>api.types.is_interval_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of the Interval dtype.
<code>api.types.is_numeric_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of a numeric dtype.
<code>api.types.is_object_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of the object dtype.
<code>api.types.is_period_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of the Period dtype.
<code>api.types.is_signed_integer_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of a signed integer dtype.
<code>api.types.is_string_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of the string dtype.
<code>api.types.is_timedelta64_dtype(arr_or_dtype)</code>	Check whether an array-like or dtype is of the timedelta64 dtype.
<code>api.types.is_timedelta64_ns_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of the timedelta64[ns] dtype.

Continued on next page

Table 185 – continued from previous page

<code>api.types.is_unsigned_integer_dtype(arr_or_dtype)</code>	Check whether the provided array or dtype is of an unsigned integer dtype.
<code>api.types.is_sparse(arr)</code>	Check whether an array-like is a pandas sparse array.

#### 34.21.4.4 pandas.api.types.is\_bool\_dtype

`pandas.api.types.is_bool_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of a boolean dtype.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of a boolean dtype.]

#### Examples

```
>>> is_bool_dtype(str)
False
>>> is_bool_dtype(int)
False
>>> is_bool_dtype(bool)
True
>>> is_bool_dtype(np.bool)
True
>>> is_bool_dtype(np.array(['a', 'b']))
False
>>> is_bool_dtype(pd.Series([1, 2]))
False
>>> is_bool_dtype(np.array([True, False]))
True
```

#### 34.21.4.5 pandas.api.types.is\_categorical\_dtype

`pandas.api.types.is_categorical_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of the Categorical dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns** **boolean** : Whether or not the array-like or dtype is of the Categorical dtype.

#### Examples

```
>>> is_categorical_dtype(object)
False
>>> is_categorical_dtype(CategoricalDtype())
True
```

(continues on next page)

(continued from previous page)

```
>>> is_categorical_dtype([1, 2, 3])
False
>>> is_categorical_dtype(pd.Categorical([1, 2, 3]))
True
>>> is_categorical_dtype(pd.CategoricalIndex([1, 2, 3]))
True
```

#### 34.21.4.6 pandas.api.types.is\_complex\_dtype

pandas.api.types.**is\_complex\_dtype**(arr\_or\_dtype)

Check whether the provided array or dtype is of a complex dtype.

**Parameters** arr\_or\_dtype : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of a complex dtype.]

#### Examples

```
>>> is_complex_dtype(str)
False
>>> is_complex_dtype(int)
False
>>> is_complex_dtype(np.complex)
True
>>> is_complex_dtype(np.array(['a', 'b']))
False
>>> is_complex_dtype(pd.Series([1, 2]))
False
>>> is_complex_dtype(np.array([1 + 1j, 5]))
True
```

#### 34.21.4.7 pandas.api.types.is\_datetime64\_any\_dtype

pandas.api.types.**is\_datetime64\_any\_dtype**(arr\_or\_dtype)

Check whether the provided array or dtype is of the datetime64 dtype.

**Parameters** arr\_or\_dtype : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of the datetime64 dtype.]

#### Examples



```

>>> is_datetime64_any_dtype(str)
False
>>> is_datetime64_any_dtype(int)
False
>>> is_datetime64_any_dtype(np.datetime64) # can be tz-naive
True
>>> is_datetime64_any_dtype(DatetimeTZDtype("ns", "US/Eastern"))
True
>>> is_datetime64_any_dtype(np.array(['a', 'b']))
False
>>> is_datetime64_any_dtype(np.array([1, 2]))
False
>>> is_datetime64_any_dtype(np.array([], dtype=np.datetime64))
True
>>> is_datetime64_any_dtype(pd.DatetimeIndex([1, 2, 3],
                                             dtype=np.datetime64))
True

```

#### 34.21.4.8 pandas.api.types.is\_datetime64\_dtype

`pandas.api.types.is_datetime64_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of the datetime64 dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns** `boolean` : Whether or not the array-like or dtype is of the datetime64 dtype.

#### Examples

```

>>> is_datetime64_dtype(object)
False
>>> is_datetime64_dtype(np.datetime64)
True
>>> is_datetime64_dtype(np.array([], dtype=int))
False
>>> is_datetime64_dtype(np.array([], dtype=np.datetime64))
True
>>> is_datetime64_dtype([1, 2, 3])
False

```

#### 34.21.4.9 pandas.api.types.is\_datetime64\_ns\_dtype

`pandas.api.types.is_datetime64_ns_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of the datetime64[ns] dtype.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns**

`boolean` [Whether or not the array or dtype is of the datetime64[ns] dtype.]

## Examples

```
>>> is_datetime64_ns_dtype(str)
False
>>> is_datetime64_ns_dtype(int)
False
>>> is_datetime64_ns_dtype(np.datetime64) # no unit
False
>>> is_datetime64_ns_dtype(DatetimeTZDtype("ns", "US/Eastern"))
True
>>> is_datetime64_ns_dtype(np.array(['a', 'b']))
False
>>> is_datetime64_ns_dtype(np.array([1, 2]))
False
>>> is_datetime64_ns_dtype(np.array([], dtype=np.datetime64)) # no unit
False
>>> is_datetime64_ns_dtype(np.array([],
                                dtype="datetime64[ps]")) # wrong unit
False
>>> is_datetime64_ns_dtype(pd.DatetimeIndex([1, 2, 3],
                                                dtype=np.datetime64)) # has 'ns' unit
True
```

### 34.21.4.10 pandas.api.types.is\_datetime64tz\_dtype

`pandas.api.types.is_datetime64tz_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of a DatetimeTZDtype dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns** `boolean` : Whether or not the array-like or dtype is of  
a DatetimeTZDtype dtype.

## Examples

```
>>> is_datetime64tz_dtype(object)
False
>>> is_datetime64tz_dtype([1, 2, 3])
False
>>> is_datetime64tz_dtype(pd.DatetimeIndex([1, 2, 3])) # tz-naive
False
>>> is_datetime64tz_dtype(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
```

```
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_datetime64tz_dtype(dtype)
True
>>> is_datetime64tz_dtype(s)
True
```

### 34.21.4.11 pandas.api.types.is\_extension\_type

`pandas.api.types.is_extension_type(arr)`

Check whether an array-like is of a pandas extension class instance.

Extension classes include categoricals, pandas sparse objects (i.e. classes represented within the pandas library and not ones external to it like scipy sparse matrices), and datetime-like arrays.

**Parameters** `arr` : array-like

The array-like to check.

**Returns** `boolean` : Whether or not the array-like is of a pandas extension class instance.

#### Examples

```
>>> is_extension_type([1, 2, 3])
False
>>> is_extension_type(np.array([1, 2, 3]))
False
>>>
>>> cat = pd.Categorical([1, 2, 3])
>>>
>>> is_extension_type(cat)
True
>>> is_extension_type(pd.Series(cat))
True
>>> is_extension_type(pd.SparseArray([1, 2, 3]))
True
>>> is_extension_type(pd.SparseSeries([1, 2, 3]))
True
>>>
>>> from scipy.sparse import bsr_matrix
>>> is_extension_type(bsr_matrix([1, 2, 3]))
False
>>> is_extension_type(pd.DatetimeIndex([1, 2, 3]))
False
>>> is_extension_type(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
>>>
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_extension_type(s)
True
```

### 34.21.4.12 pandas.api.types.is\_float\_dtype

`pandas.api.types.is_float_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of a float dtype.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of a float dtype.]

### Examples

```
>>> is_float_dtype(str)
False
>>> is_float_dtype(int)
False
>>> is_float_dtype(float)
True
>>> is_float_dtype(np.array(['a', 'b']))
False
>>> is_float_dtype(pd.Series([1, 2]))
False
>>> is_float_dtype(pd.Index([1, 2.]))
True
```

#### 34.21.4.13 pandas.api.types.is\_int64\_dtype

pandas.api.types.**is\_int64\_dtype**(*arr\_or\_dtype*)

Check whether the provided array or dtype is of the int64 dtype.

**Parameters** *arr\_or\_dtype* : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of the int64 dtype.]

### Notes

Depending on system architecture, the return value of *is\_int64\_dtype*(*int*) will be True if the OS uses 64-bit integers and False if the OS uses 32-bit integers.

### Examples

```
>>> is_int64_dtype(str)
False
>>> is_int64_dtype(np.int32)
False
>>> is_int64_dtype(np.int64)
True
>>> is_int64_dtype(float)
False
>>> is_int64_dtype(np.uint64) # unsigned
False
>>> is_int64_dtype(np.array(['a', 'b']))
False
>>> is_int64_dtype(np.array([1, 2], dtype=np.int64))
True
>>> is_int64_dtype(pd.Index([1, 2.])) # float
False
```

(continues on next page)

(continued from previous page)

```
>>> is_int64_dtype(np.array([1, 2], dtype=np.uint32)) # unsigned
False
```

#### 34.21.4.14 pandas.api.types.is\_integer\_dtype

`pandas.api.types.is_integer_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of an integer dtype.

Unlike in *in\_any\_int\_dtype*, *timedelta64* instances will return False.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns** `boolean` : Whether or not the array or dtype is of an integer dtype and not an instance of *timedelta64*.

#### Examples

```
>>> is_integer_dtype(str)
False
>>> is_integer_dtype(int)
True
>>> is_integer_dtype(float)
False
>>> is_integer_dtype(np.uint64)
True
>>> is_integer_dtype(np.datetime64)
False
>>> is_integer_dtype(np.timedelta64)
False
>>> is_integer_dtype(np.array(['a', 'b']))
False
>>> is_integer_dtype(pd.Series([1, 2]))
True
>>> is_integer_dtype(np.array([], dtype=np.timedelta64))
False
>>> is_integer_dtype(pd.Index([1, 2.])) # float
False
```

#### 34.21.4.15 pandas.api.types.is\_interval\_dtype

`pandas.api.types.is_interval_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of the Interval dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns** `boolean` : Whether or not the array-like or dtype is of the Interval dtype.

## Examples

```
>>> is_interval_dtype(object)
False
>>> is_interval_dtype(IntervalDtype())
True
>>> is_interval_dtype([1, 2, 3])
False
>>>
>>> interval = pd.Interval(1, 2, closed="right")
>>> is_interval_dtype(interval)
False
>>> is_interval_dtype(pd.IntervalIndex([interval]))
True
```

### 34.21.4.16 pandas.api.types.is\_numeric\_dtype

pandas.api.types.is\_numeric\_dtype(arr\_or\_dtype)

Check whether the provided array or dtype is of a numeric dtype.

**Parameters** arr\_or\_dtype : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of a numeric dtype.]

## Examples

```
>>> is_numeric_dtype(str)
False
>>> is_numeric_dtype(int)
True
>>> is_numeric_dtype(float)
True
>>> is_numeric_dtype(np.uint64)
True
>>> is_numeric_dtype(np.datetime64)
False
>>> is_numeric_dtype(np.timedelta64)
False
>>> is_numeric_dtype(np.array(['a', 'b']))
False
>>> is_numeric_dtype(pd.Series([1, 2]))
True
>>> is_numeric_dtype(pd.Index([1, 2.]))
True
>>> is_numeric_dtype(np.array([], dtype=np.timedelta64))
False
```

### 34.21.4.17 pandas.api.types.is\_object\_dtype

pandas.api.types.is\_object\_dtype(arr\_or\_dtype)

Check whether an array-like or dtype is of the object dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns**

**boolean** [Whether or not the array-like or dtype is of the object dtype.]

### Examples

```
>>> is_object_dtype(object)
True
>>> is_object_dtype(int)
False
>>> is_object_dtype(np.array([], dtype=object))
True
>>> is_object_dtype(np.array([], dtype=int))
False
>>> is_object_dtype([1, 2, 3])
False
```

#### 34.21.4.18 pandas.api.types.is\_period\_dtype

`pandas.api.types.is_period_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of the Period dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns**

**boolean** [Whether or not the array-like or dtype is of the Period dtype.]

### Examples

```
>>> is_period_dtype(object)
False
>>> is_period_dtype(PeriodDtype(freq="D"))
True
>>> is_period_dtype([1, 2, 3])
False
>>> is_period_dtype(pd.Period("2017-01-01"))
False
>>> is_period_dtype(pd.PeriodIndex([], freq="A"))
True
```

#### 34.21.4.19 pandas.api.types.is\_signed\_integer\_dtype

`pandas.api.types.is_signed_integer_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of a signed integer dtype.

Unlike in `in_any_int_dtype`, `timedelta64` instances will return False.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns** **boolean** : Whether or not the array or dtype is of a signed integer dtype and not an instance of timedelta64.

### Examples

```
>>> is_signed_integer_dtype(str)
False
>>> is_signed_integer_dtype(int)
True
>>> is_signed_integer_dtype(float)
False
>>> is_signed_integer_dtype(np.uint64)  # unsigned
False
>>> is_signed_integer_dtype(np.datetime64)
False
>>> is_signed_integer_dtype(np.timedelta64)
False
>>> is_signed_integer_dtype(np.array(['a', 'b']))
False
>>> is_signed_integer_dtype(pd.Series([1, 2]))
True
>>> is_signed_integer_dtype(np.array([], dtype=np.timedelta64))
False
>>> is_signed_integer_dtype(pd.Index([1, 2.]))  # float
False
>>> is_signed_integer_dtype(np.array([1, 2], dtype=np.uint32))  # unsigned
False
```

#### 34.21.4.20 pandas.api.types.is\_string\_dtype

`pandas.api.types.is_string_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of the string dtype.

**Parameters** **arr\_or\_dtype** : array-like

The array or dtype to check.

**Returns**

**boolean** [Whether or not the array or dtype is of the string dtype.]

### Examples

```
>>> is_string_dtype(str)
True
>>> is_string_dtype(object)
True
>>> is_string_dtype(int)
False
>>>
>>> is_string_dtype(np.array(['a', 'b']))
True
```

(continues on next page)



(continued from previous page)

```
>>> is_string_dtype(pd.Series([1, 2]))
False
```

#### 34.21.4.21 pandas.api.types.is\_timedelta64\_dtype

`pandas.api.types.is_timedelta64_dtype(arr_or_dtype)`

Check whether an array-like or dtype is of the timedelta64 dtype.

**Parameters** `arr_or_dtype` : array-like

The array-like or dtype to check.

**Returns** `boolean` : Whether or not the array-like or dtype is of the timedelta64 dtype.

#### Examples

```
>>> is_timedelta64_dtype(object)
False
>>> is_timedelta64_dtype(np.timedelta64)
True
>>> is_timedelta64_dtype([1, 2, 3])
False
>>> is_timedelta64_dtype(pd.Series([], dtype="timedelta64[ns]"))
True
>>> is_timedelta64_dtype('0 days')
False
```

#### 34.21.4.22 pandas.api.types.is\_timedelta64\_ns\_dtype

`pandas.api.types.is_timedelta64_ns_dtype(arr_or_dtype)`

Check whether the provided array or dtype is of the timedelta64[ns] dtype.

This is a very specific dtype, so generic ones like `np.timedelta64` will return False if passed into this function.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns** `boolean` : Whether or not the array or dtype is of the timedelta64[ns] dtype.

#### Examples

```
>>> is_timedelta64_ns_dtype(np.dtype('m8[ns]'))
True
>>> is_timedelta64_ns_dtype(np.dtype('m8[ps]')) # Wrong frequency
False
>>> is_timedelta64_ns_dtype(np.array([1, 2], dtype='m8[ns]'))
True
>>> is_timedelta64_ns_dtype(np.array([1, 2], dtype=np.timedelta64))
False
```

#### 34.21.4.23 pandas.api.types.is\_unsigned\_integer\_dtype

`pandas.api.types.is_unsigned_integer_dtype(arr_or_dtype)`  
Check whether the provided array or dtype is of an unsigned integer dtype.

**Parameters** `arr_or_dtype` : array-like

The array or dtype to check.

**Returns** `boolean` : Whether or not the array or dtype is of an unsigned integer dtype.

##### Examples

```
>>> is_unsigned_integer_dtype(str)
False
>>> is_unsigned_integer_dtype(int)    # signed
False
>>> is_unsigned_integer_dtype(float)
False
>>> is_unsigned_integer_dtype(np.uint64)
True
>>> is_unsigned_integer_dtype(np.array(['a', 'b']))
False
>>> is_unsigned_integer_dtype(pd.Series([1, 2]))    # signed
False
>>> is_unsigned_integer_dtype(pd.Index([1, 2]))    # float
False
>>> is_unsigned_integer_dtype(np.array([1, 2], dtype=np.uint32))
True
```

#### 34.21.4.24 pandas.api.types.is\_sparse

`pandas.api.types.is_sparse(arr)`  
Check whether an array-like is a pandas sparse array.

**Parameters** `arr` : array-like

The array-like to check.

**Returns**

`boolean` [Whether or not the array-like is a pandas sparse array.]

##### Examples

```
>>> is_sparse(np.array([1, 2, 3]))
False
>>> is_sparse(pd.SparseArray([1, 2, 3]))
True
>>> is_sparse(pd.SparseSeries([1, 2, 3]))
True
```

This function checks only for pandas sparse array instances, so sparse arrays from other libraries will return False.

```
>>> from scipy.sparse import bsr_matrix
>>> is_sparse(bsr_matrix([1, 2, 3]))
False
```

Iterable introspection

<code>api.types.is_dict_like(obj)</code>	Check if the object is dict-like.
<code>api.types.is_file_like(obj)</code>	Check if the object is a file-like object.
<code>api.types.is_list_like(obj)</code>	Check if the object is list-like.
<code>api.types.is_named_tuple(obj)</code>	Check if the object is a named tuple.
<code>api.types.is_iterator(obj)</code>	Check if the object is an iterator.

#### 34.21.4.25 pandas.api.types.is\_dict\_like

`pandas.api.types.is_dict_like(obj)`

Check if the object is dict-like.

##### Parameters

**obj** [The object to check.]

**Returns** `is_dict_like` : bool

Whether *obj* has dict-like properties.

##### Examples

```
>>> is_dict_like({1: 2})
True
>>> is_dict_like([1, 2, 3])
False
```

#### 34.21.4.26 pandas.api.types.is\_file\_like

`pandas.api.types.is_file_like(obj)`

Check if the object is a file-like object.

For objects to be considered file-like, they must be an iterator AND have either a *read* and/or *write* method as an attribute.

Note: file-like objects must be iterable, but iterable objects need not be file-like.

New in version 0.20.0.

##### Parameters

**obj** [The object to check.]

**Returns** `is_file_like` : bool

Whether *obj* has file-like properties.

## Examples

```
>>> buffer(StringIO("data"))
>>> is_file_like(buffer)
True
>>> is_file_like([1, 2, 3])
False
```

### 34.21.4.27 pandas.api.types.is\_list\_like

`pandas.api.types.is_list_like(obj)`

Check if the object is list-like.

Objects that are considered list-like are for example Python lists, tuples, sets, NumPy arrays, and Pandas Series.

Strings and datetime objects, however, are not considered list-like.

#### Parameters

**obj** [The object to check.]

**Returns** `is_list_like` : bool

Whether *obj* has list-like properties.

## Examples

```
>>> is_list_like([1, 2, 3])
True
>>> is_list_like({1, 2, 3})
True
>>> is_list_like(datetime(2017, 1, 1))
False
>>> is_list_like("foo")
False
>>> is_list_like(1)
False
```

### 34.21.4.28 pandas.api.types.is\_named\_tuple

`pandas.api.types.is_named_tuple(obj)`

Check if the object is a named tuple.

#### Parameters

**obj** [The object to check.]

**Returns** `is_named_tuple` : bool

Whether *obj* is a named tuple.

## Examples

```

>>> Point = namedtuple("Point", ["x", "y"])
>>> p = Point(1, 2)
>>>
>>> is_named_tuple(p)
True
>>> is_named_tuple((1, 2))
False

```

#### 34.21.4.29 pandas.api.types.is\_iterator

`pandas.api.types.is_iterator(obj)`

Check if the object is an iterator.

For example, lists are considered iterators but not strings or datetime objects.

##### Parameters

**obj** [The object to check.]

**Returns** `is_iter` : bool

Whether *obj* is an iterator.

##### Examples

```

>>> is_iterator([1, 2, 3])
True
>>> is_iterator(datetime(2017, 1, 1))
False
>>> is_iterator("foo")
False
>>> is_iterator(1)
False

```

Scalar introspection

<code>api.types.is_bool</code>	
<code>api.types.is_categorical(arr)</code>	Check whether an array-like is a Categorical instance.
<code>api.types.is_complex</code>	
<code>api.types.is_datetimetz(arr)</code>	Check whether an array-like is a datetime array-like with a timezone component in its dtype.
<code>api.types.is_float</code>	
<code>api.types.is_hashable(obj)</code>	Return True if hash(obj) will succeed, False otherwise.
<code>api.types.is_integer</code>	
<code>api.types.is_interval</code>	
<code>api.types.is_number(obj)</code>	Check if the object is a number.
<code>api.types.is_period(arr)</code>	Check whether an array-like is a periodical index.
<code>api.types.is_re(obj)</code>	Check if the object is a regex pattern instance.
<code>api.types.is_re_compilable(obj)</code>	Check if the object can be compiled into a regex pattern instance.
<code>api.types.is_scalar</code>	Return True if given value is scalar.

#### 34.21.4.30 pandas.api.types.is\_bool

`pandas.api.types.is_bool()`

#### 34.21.4.31 pandas.api.types.is\_categorical

`pandas.api.types.is_categorical(arr)`

Check whether an array-like is a Categorical instance.

**Parameters** `arr`: array-like

The array-like to check.

**Returns**

**boolean** [Whether or not the array-like is of a Categorical instance.]

#### Examples

```
>>> is_categorical([1, 2, 3])
False
```

Categoricals, Series Categoricals, and CategoricalIndex will return True.

```
>>> cat = pd.Categorical([1, 2, 3])
>>> is_categorical(cat)
True
>>> is_categorical(pd.Series(cat))
True
>>> is_categorical(pd.CategoricalIndex([1, 2, 3]))
True
```

#### 34.21.4.32 pandas.api.types.is\_complex

`pandas.api.types.is_complex()`

#### 34.21.4.33 pandas.api.types.is\_datetimetz

`pandas.api.types.is_datetimetz(arr)`

Check whether an array-like is a datetime array-like with a timezone component in its dtype.

**Parameters** `arr`: array-like

The array-like to check.

**Returns** **boolean**: Whether or not the array-like is a datetime array-like with a timezone component in its dtype.

#### Examples

```
>>> is_datetimetz([1, 2, 3])
False
```

Although the following examples are both `DatetimeIndex` objects, the first one returns `False` because it has no timezone component unlike the second one, which returns `True`.

```
>>> is_datetimez(pd.DatetimeIndex([1, 2, 3]))
False
>>> is_datetimez(pd.DatetimeIndex([1, 2, 3], tz="US/Eastern"))
True
```

The object need not be a `DatetimeIndex` object. It just needs to have a `dtype` which has a timezone component.

```
>>> dtype = DatetimeTZDtype("ns", tz="US/Eastern")
>>> s = pd.Series([], dtype=dtype)
>>> is_datetimez(s)
True
```

#### 34.21.4.34 pandas.api.types.is\_float

`pandas.api.types.is_float()`

#### 34.21.4.35 pandas.api.types.is\_hashable

`pandas.api.types.is_hashable(obj)`

Return `True` if `hash(obj)` will succeed, `False` otherwise.

Some types will pass a test against `collections.Hashable` but fail when they are actually hashed with `hash()`.

Distinguish between these and other types by trying the call to `hash()` and seeing if they raise `TypeError`.

#### Examples

```
>>> a = ([],)
>>> isinstance(a, collections.Hashable)
True
>>> is_hashable(a)
False
```

#### 34.21.4.36 pandas.api.types.is\_integer

`pandas.api.types.is_integer()`

#### 34.21.4.37 pandas.api.types.is\_interval

`pandas.api.types.is_interval()`

#### 34.21.4.38 pandas.api.types.is\_number

`pandas.api.types.is_number(obj)`

Check if the object is a number.

Returns `True` when the object is a number, and `False` if is not.

**Parameters** `obj`: any type

The object to check if is a number.

**Returns** `is_number` : bool

Whether *obj* is a number or not.

**See also:**

`pandas.api.types.is_integer` checks a subgroup of numbers

### Examples

```
>>> pd.api.types.is_number(1)
True
>>> pd.api.types.is_number(7.15)
True
```

Booleans are valid because they are int subclass.

```
>>> pd.api.types.is_number(False)
True
```

```
>>> pd.api.types.is_number("foo")
False
>>> pd.api.types.is_number("5")
False
```

#### 34.21.4.39 pandas.api.types.is\_period

`pandas.api.types.is_period(arr)`

Check whether an array-like is a periodical index.

**Parameters** `arr` : array-like

The array-like to check.

**Returns**

**boolean** [Whether or not the array-like is a periodical index.]

### Examples

```
>>> is_period([1, 2, 3])
False
>>> is_period(pd.Index([1, 2, 3]))
False
>>> is_period(pd.PeriodIndex(["2017-01-01"], freq="D"))
True
```

#### 34.21.4.40 pandas.api.types.is\_re

`pandas.api.types.is_re(obj)`

Check if the object is a regex pattern instance.



**Parameters****obj** [The object to check.]**Returns** **is\_regex** : boolWhether *obj* is a regex pattern.**Examples**

```
>>> is_re(re.compile(".*"))
True
>>> is_re("foo")
False
```

**34.21.4.41 pandas.api.types.is\_re\_compilable****pandas.api.types.is\_re\_compilable** (*obj*)

Check if the object can be compiled into a regex pattern instance.

**Parameters****obj** [The object to check.]**Returns** **is\_regex\_compilable** : boolWhether *obj* can be compiled as a regex pattern.**Examples**

```
>>> is_re_compilable(".*")
True
>>> is_re_compilable(1)
False
```

**34.21.4.42 pandas.api.types.is\_scalar****pandas.api.types.is\_scalar** ()

Return True if given value is scalar.

This includes: - numpy array scalar (e.g. np.int64) - Python builtin numerics - Python builtin byte arrays and strings - None - instances of datetime.datetime - instances of datetime.timedelta - Period - instances of decimal.Decimal - Interval - DateOffset

**34.22 Extensions**

These are primarily intended for library authors looking to extend pandas objects.

---

`api.extensions.register_dataframe_accessor(name)` Register a custom accessor on DataFrame objects.

---

`api.extensions.register_series_accessor(name)` Register a custom accessor on Series objects.

---

`api.extensions.register_index_accessor(name)` Register a custom accessor on Index objects.

Continued on next page

Table 188 – continued from previous page

<code>api.extensions.ExtensionDtype</code>	A custom data type, to be paired with an ExtensionArray.
<code>api.extensions.ExtensionArray</code>	Abstract base class for custom 1-D array types.

### 34.22.1 pandas.api.extensions.register\_dataframe\_accessor

`pandas.api.extensions.register_dataframe_accessor` (*name*)

Register a custom accessor on DataFrame objects.

**Parameters** *name* : str

Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

**See also:**

`register_series_accessor`, `register_index_accessor`

#### Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```
>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values
```

#### Examples

In your library code:

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
```

(continues on next page)

(continued from previous page)

```
# plot this array's data on a map, e.g., using Cartopy
pass
```

Back in an interactive IPython session:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

### 34.22.2 pandas.api.extensions.register\_series\_accessor

pandas.api.extensions.**register\_series\_accessor**(*name*)

Register a custom accessor on Series objects.

**Parameters** *name* : str

Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

**See also:**

*register\_dataframe\_accessor, register\_index\_accessor*

#### Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```
>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values
```

#### Examples

In your library code:

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
```

(continues on next page)

(continued from previous page)

```

def center(self):
    # return the geographic center point of this DataFrame
    lat = self._obj.latitude
    lon = self._obj.longitude
    return (float(lon.mean()), float(lat.mean()))

def plot(self):
    # plot this array's data on a map, e.g., using Cartopy
    pass

```

Back in an interactive IPython session:

```

>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map

```

### 34.22.3 pandas.api.extensions.register\_index\_accessor

pandas.api.extensions.register\_index\_accessor(*name*)

Register a custom accessor on Index objects.

**Parameters** *name* : str

Name under which the accessor should be registered. A warning is issued if this name conflicts with a preexisting attribute.

**See also:**

*register\_dataframe\_accessor, register\_series\_accessor*

#### Notes

When accessed, your accessor will be initialized with the pandas object the user is interacting with. So the signature must be

```
def __init__(self, pandas_object):
```

For consistency with pandas methods, you should raise an `AttributeError` if the data passed to your accessor has an incorrect dtype.

```

>>> pd.Series(['a', 'b']).dt
Traceback (most recent call last):
...
AttributeError: Can only use .dt accessor with datetimelike values

```

#### Examples

In your library code:

```
import pandas as pd

@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Back in an interactive IPython session:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

### 34.22.4 pandas.api.extensions.ExtensionDtype

**class** pandas.api.extensions.**ExtensionDtype**

A custom data type, to be paired with an ExtensionArray.

New in version 0.23.0.

#### Notes

The interface includes the following abstract methods that must be implemented by subclasses:

- type
- name
- construct\_from\_string

The *na\_value* class attribute can be used to set the default NA value for this type. `numpy.nan` is used by default.

This class does not inherit from ‘abc.ABCMeta’ for performance reasons. Methods and properties required by the interface raise `pandas.errors.AbstractMethodError` and no `register` method is provided for registering virtual subclasses.

#### Attributes

<i>kind</i>	A character code (one of 'biufcmMOSUV'), default 'O'
<i>name</i>	A string identifying the data type.
<i>names</i>	Ordered list of field names, or None if there are no fields.
<i>type</i>	The scalar type for the array, e.g.

#### 34.22.4.1 pandas.api.extensions.ExtensionDtype.kind

`ExtensionDtype.kind`

A character code (one of 'biufcmMOSUV'), default 'O'

This should match the NumPy dtype used when the array is converted to an ndarray, which is probably 'O' for object if the extension type cannot be represented as a built-in NumPy type.

**See also:**

`numpy.dtype.kind`

#### 34.22.4.2 pandas.api.extensions.ExtensionDtype.name

`ExtensionDtype.name`

A string identifying the data type.

Will be used for display in, e.g. `Series.dtype`

#### 34.22.4.3 pandas.api.extensions.ExtensionDtype.names

`ExtensionDtype.names`

Ordered list of field names, or None if there are no fields.

This is for compatibility with NumPy arrays, and may be removed in the future.

#### 34.22.4.4 pandas.api.extensions.ExtensionDtype.type

`ExtensionDtype.type`

The scalar type for the array, e.g. `int`

It's expected `ExtensionArray[item]` returns an instance of `ExtensionDtype.type` for scalar `item`.

### Methods

<i><code>construct_from_string</code></i> (string)	Attempt to construct this type from a string.
<i><code>is_dtype</code></i> (dtype)	Check if we match 'dtype'.

#### 34.22.4.5 pandas.api.extensions.ExtensionDtype.construct\_from\_string

**classmethod** `ExtensionDtype.construct_from_string(string)`

Attempt to construct this type from a string.

**Parameters**

**string** [str]

**Returns**

**self** [instance of 'cls']

**Raises** **TypeError**

If a class cannot be constructed from this 'string'.

**Examples**

If the extension dtype can be constructed without any arguments, the following may be an adequate implementation.

```
>>> @classmethod
... def construct_from_string(cls, string)
...     if string == cls.name:
...         return cls()
...     else:
...         raise TypeError("Cannot construct a '{}' from "
...                        "'{}'.format(cls, string))
...
```

#### 34.22.4.6 pandas.api.extensions.ExtensionDtype.is\_dtype

**classmethod** `ExtensionDtype.is_dtype(dtype)`

Check if we match 'dtype'.

**Parameters** `dtype` : object

The object to check.

**Returns**

**is\_dtype** [bool]

**Notes**

The default implementation is True if

1. `cls.construct_from_string(dtype)` is an instance of `cls`.
2. `dtype` is an object and is an instance of `cls`
3. `dtype` has a `dtype` attribute, and any of the above conditions is true for `dtype.dtype`.

#### 34.22.5 pandas.api.extensions.ExtensionArray

**class** `pandas.api.extensions.ExtensionArray`

Abstract base class for custom 1-D array types.

pandas will recognize instances of this class as proper arrays with a custom type and will not attempt to coerce them to objects. They may be stored directly inside a `DataFrame` or `Series`.

New in version 0.23.0.

## Notes

The interface includes the following abstract methods that must be implemented by subclasses:

- `_from_sequence`
- `_from_factorized`
- `__getitem__`
- `__len__`
- `dtype`
- `nbytes`
- `isna`
- `take`
- `copy`
- `_concat_same_type`

An additional method is available to satisfy pandas' internal, private block API.

- `_formatting_values`

Some methods require casting the `ExtensionArray` to an `ndarray` of Python objects with `self.astype(object)`, which may be expensive. When performance is a concern, we highly recommend overriding the following methods:

- `fillna`
- `unique`
- `factorize / _values_for_factorize`
- `argsort / _values_for_argsort`

This class does not inherit from `'abc.ABCMeta'` for performance reasons. Methods and properties required by the interface raise `pandas.errors.AbstractMethodError` and no `register` method is provided for registering virtual subclasses.

`ExtensionArrays` are limited to 1 dimension.

They may be backed by none, one, or many NumPy arrays. For example, `pandas.Categorical` is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 address may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists. Pandas makes no assumptions on how the data are stored, just that it can be converted to a NumPy array. The `ExtensionArray` interface does not impose any rules on how this data is stored. However, currently, the backing data cannot be stored in attributes called `.values` or `._values` to ensure full compatibility with pandas internals. But other names as `.data`, `._data`, `._items`, ... can be freely used.

## Attributes

<i>dtype</i>	An instance of <code>'ExtensionDtype'</code> .
<i>nbytes</i>	The number of bytes needed to store this object in memory.

Continued on next page



Table 191 – continued from previous page

<i>ndim</i>	Extension Arrays are only allowed to be 1-dimensional.
<i>shape</i>	Return a tuple of the array dimensions.

**34.22.5.1 pandas.api.extensions.ExtensionArray.dtype**`ExtensionArray.dtype`

An instance of ‘ExtensionDtype’.

**34.22.5.2 pandas.api.extensions.ExtensionArray.nbytes**`ExtensionArray.nbytes`

The number of bytes needed to store this object in memory.

**34.22.5.3 pandas.api.extensions.ExtensionArray.ndim**`ExtensionArray.ndim`

Extension Arrays are only allowed to be 1-dimensional.

**34.22.5.4 pandas.api.extensions.ExtensionArray.shape**`ExtensionArray.shape`

Return a tuple of the array dimensions.

**Methods**

<i>argsort</i> ([ascending, kind])	Return the indices that would sort this array.
<i>astype</i> (dtype[, copy])	Cast to a NumPy array with ‘dtype’.
<i>copy</i> ([deep])	Return a copy of the array.
<i>factorize</i> ([na_sentinel])	Encode the extension array as an enumerated type.
<i>fillna</i> ([value, method, limit])	Fill NA/NaN values using the specified method.
<i>isna</i> ()	Boolean NumPy array indicating if each value is missing.
<i>take</i> (indices[, allow_fill, fill_value])	Take elements from an array.
<i>unique</i> ()	Compute the ExtensionArray of unique values.

**34.22.5.5 pandas.api.extensions.ExtensionArray.argsort**`ExtensionArray.argsort` (*ascending=True*, *kind='quicksort'*, *\*args*, *\*\*kwargs*)

Return the indices that would sort this array.

**Parameters** *ascending* : bool, default True

Whether the indices should result in an ascending or descending sort.

**kind** : { ‘quicksort’, ‘mergesort’, ‘heapsort’ }, optional

Sorting algorithm.

**\*args, \*\*kwargs**:

passed through to `numpy.argsort()`.

**Returns** `index_array` : ndarray

Array of indices that sort `self`.

**See also:**

`numpy.argsort` Sorting implementation used internally.

#### 34.22.5.6 `pandas.api.extensions.ExtensionArray.astype`

`ExtensionArray.astype(dtype, copy=True)`

Cast to a NumPy array with 'dtype'.

**Parameters** `dtype` : str or dtype

Typecode or data-type to which the array is cast.

**copy** : bool, default True

Whether to copy the data, even if not necessary. If False, a copy is made only if the old dtype does not match the new dtype.

**Returns** `array` : ndarray

NumPy ndarray with 'dtype' for its dtype.

#### 34.22.5.7 `pandas.api.extensions.ExtensionArray.copy`

`ExtensionArray.copy(deep=False)`

Return a copy of the array.

**Parameters** `deep` : bool, default False

Also copy the underlying data backing this array.

**Returns**

`ExtensionArray`

#### 34.22.5.8 `pandas.api.extensions.ExtensionArray.factorize`

`ExtensionArray.factorize(na_sentinel=-1)`

Encode the extension array as an enumerated type.

**Parameters** `na_sentinel` : int, default -1

Value to use in the `labels` array to indicate missing values.

**Returns** `labels` : ndarray

An integer NumPy array that's an indexer into the original `ExtensionArray`.

**uniques** : `ExtensionArray`

An `ExtensionArray` containing the unique values of `self`.

---

**Note:** `uniques` will *not* contain an entry for the NA value of the `ExtensionArray` if there are any missing values present in `self`.

---

See also:

`pandas.factorize` Top-level factorize method that dispatches here.

## Notes

`pandas.factorize()` offers a *sort* keyword as well.

### 34.22.5.9 pandas.api.extensions.ExtensionArray.fillna

`ExtensionArray.fillna(value=None, method=None, limit=None)`

Fill NA/NaN values using the specified method.

**Parameters** **value** : scalar, array-like

If a scalar value is passed it is used to fill all missing values. Alternatively, an array-like ‘value’ can be given. It’s expected that the array-like have the same length as ‘self’.

**method** : {‘backfill’, ‘bfill’, ‘pad’, ‘ffill’, None}, default None

Method to use for filling holes in reindexed Series pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation to fill gap

**limit** : int, default None

If method is specified, this is the maximum number of consecutive NaN values to forward/backward fill. In other words, if there is a gap with more than this number of consecutive NaNs, it will only be partially filled. If method is not specified, this is the maximum number of entries along the entire axis where NaNs will be filled.

## Returns

**filled** [ExtensionArray with NA/NaN filled]

### 34.22.5.10 pandas.api.extensions.ExtensionArray.isna

`ExtensionArray.isna()`

Boolean NumPy array indicating if each value is missing.

This should return a 1-D array the same length as ‘self’.

### 34.22.5.11 pandas.api.extensions.ExtensionArray.take

`ExtensionArray.take(indices, allow_fill=False, fill_value=None)`

Take elements from an array.

**Parameters** **indices** : sequence of integers

Indices to be taken.

**allow\_fill** : bool, default False

How to handle negative values in *indices*.

- False: negative values in *indices* indicate positional indices from the right (the default). This is similar to `numpy.take()`.
- True: negative values in *indices* indicate missing values. These values are set to *fill\_value*. Any other other negative values raise a `ValueError`.

**fill\_value** : any, optional

Fill value to use for NA-indices when *allow\_fill* is True. This may be `None`, in which case the default NA value for the type, `self.dtype.na_value`, is used.

For many `ExtensionArrays`, there will be two representations of *fill\_value*: a user-facing “boxed” scalar, and a low-level physical NA value. *fill\_value* should be the user-facing version, and the implementation should handle translating that to the physical version for processing the take if necessary.

### Returns

**ExtensionArray**

**Raises** `IndexError`

When the indices are out of bounds for the array.

**ValueError**

When *indices* contains negative values other than `-1` and *allow\_fill* is True.

### See also:

`numpy.take`, `pandas.api.extensions.take`

### Notes

`ExtensionArray.take` is called by `Series.__getitem__`, `.loc`, `iloc`, when *indices* is a sequence of values. Additionally, it's called by `Series.reindex()`, or any other method that causes realignment, with a *fill\_value*.

### Examples

Here's an example implementation, which relies on casting the extension array to object dtype. This uses the helper method `pandas.api.extensions.take()`.

```
def take(self, indices, allow_fill=False, fill_value=None):
    from pandas.core.algorithms import take

    # If the ExtensionArray is backed by an ndarray, then
    # just pass that here instead of coercing to object.
    data = self.astype(object)

    if allow_fill and fill_value is None:
        fill_value = self.dtype.na_value

    # fill value should always be translated from the scalar
    # type for the array, to the physical storage type for
    # the data, before passing to take.

    result = take(data, indices, fill_value=fill_value,
```

(continues on next page)

(continued from previous page)

```
        allow_fill=allow_fill)
    return self._from_sequence(result)
```

#### 34.22.5.12 pandas.api.extensions.ExtensionArray.unique

`ExtensionArray.unique()`

Compute the ExtensionArray of unique values.

**Returns**

`uniques` [ExtensionArray]

#### 34.22.6 pandas.Index.asi8

`Index.asi8 = None`

#### 34.22.7 pandas.Index.holds\_integer

`Index.holds_integer()`

#### 34.22.8 pandas.Index.is\_type\_compatible

`Index.is_type_compatible(kind)`

#### 34.22.9 pandas.Index.nlevels

`Index.nlevels`

#### 34.22.10 pandas.Index.sort

`Index.sort(*args, **kwargs)`

#### 34.22.11 pandas.Panel.agg

`Panel.agg(func, *args, **kwargs)`

#### 34.22.12 pandas.Panel.aggregate

`Panel.aggregate(func, *args, **kwargs)`

#### 34.22.13 pandas.Panel.is\_copy

`Panel.is_copy`

### 34.22.14 pandas.Series.imag

`Series.imag`

### 34.22.15 pandas.Series.real

`Series.real`

This section will focus on downstream applications of pandas.

## 35.1 Storing pandas DataFrame objects in Apache Parquet format

The [Apache Parquet](#) format provides key-value metadata at the file and column level, stored in the footer of the Parquet file:

```
5: optional list<KeyValue> key_value_metadata
```

where KeyValue is

```
struct KeyValue {  
  1: required string key  
  2: optional string value  
}
```

So that a `pandas.DataFrame` can be faithfully reconstructed, we store a pandas metadata key in the `FileMetaData` with the value stored as :

```
{'index_columns': ['__index_level_0__', '__index_level_1__', ...],  
'column_indexes': [<ci0>, <ci1>, ..., <ciN>],  
'columns': [<c0>, <c1>, ...],  
'pandas_version': $VERSION}
```

Here, `<c0>/<ci0>` and so forth are dictionaries containing the metadata for each column. This has JSON form:

```
{'name': column_name,  
'pandas_type': pandas_type,  
'numpy_type': numpy_type,  
'metadata': metadata}
```

`pandas_type` is the logical type of the column, and is one of:

- Boolean: 'bool'
- Integers: 'int8', 'int16', 'int32', 'int64', 'uint8', 'uint16', 'uint32', 'uint64'
- Floats: 'float16', 'float32', 'float64'
- Date and Time Types: 'datetime', 'datetimeetz', 'timedelta'
- String: 'unicode', 'bytes'

- Categorical: 'categorical'
- Other Python objects: 'object'

The `numpy_type` is the physical storage type of the column, which is the result of `str(dtype)` for the underlying NumPy array that holds the data. So for `datetime64[ns]` this is `datetime64[ns]` and for categorical, it may be any of the supported integer categorical types.

The `metadata` field is `None` except for:

- `datetime64[ns]`: {'timezone': zone, 'unit': 'ns'}, e.g. {'timezone': 'America/New\_York', 'unit': 'ns'}. The 'unit' is optional, and if omitted it is assumed to be nanoseconds.
  - categorical: {'num\_categories': K, 'ordered': is\_ordered, 'type': \$TYPE}
    - Here 'type' is optional, and can be a nested pandas type specification here (but not categorical)
  - unicode: {'encoding': encoding}
    - The encoding is optional, and if not present is UTF-8
  - object: {'encoding': encoding}. Objects can be serialized and stored in `BYTE_ARRAY` Parquet columns. The encoding can be one of:
    - 'pickle'
    - 'msgpack'
    - 'bson'
    - 'json'
  - `timedelta64[ns]`: {'unit': 'ns'}. The 'unit' is optional, and if omitted it is assumed to be nanoseconds.
- This metadata is optional altogether

For types other than these, the 'metadata' key can be omitted. Implementations can assume `None` if the key is not present.

As an example of fully-formed metadata:

```
{'index_columns': ['__index_level_0__'],
 'column_indexes': [
   {'name': None,
    'pandas_type': 'string',
    'numpy_type': 'object',
    'metadata': None}
 ],
 'columns': [
   {'name': 'c0',
    'pandas_type': 'int8',
    'numpy_type': 'int8',
    'metadata': None},
   {'name': 'c1',
    'pandas_type': 'bytes',
    'numpy_type': 'object',
    'metadata': None},
   {'name': 'c2',
    'pandas_type': 'categorical',
    'numpy_type': 'int16',
    'metadata': {'num_categories': 1000, 'ordered': False}},
   {'name': 'c3',
    'pandas_type': 'datetime64[ns]',
    'numpy_type': 'datetime64[ns]',
```

(continues on next page)



(continued from previous page)

```
    'metadata': {'timezone': 'America/Los_Angeles'}},  
    {'name': 'c4',  
      'pandas_type': 'object',  
      'numpy_type': 'object',  
      'metadata': {'encoding': 'pickle'}},  
    {'name': '__index_level_0__',  
      'pandas_type': 'int64',  
      'numpy_type': 'int64',  
      'metadata': None}  
  ],  
  'pandas_version': '0.20.0'}
```



## INTERNALS

This section will provide a look into some of pandas internals. It's primarily intended for developers of pandas itself.

### 36.1 Indexing

In pandas there are a few objects implemented which can serve as valid containers for the axis labels:

- `Index`: the generic “ordered set” object, an ndarray of object dtype assuming nothing about its contents. The labels must be hashable (and likely immutable) and unique. Populates a dict of label to location in Cython to do  $O(1)$  lookups.
- `Int64Index`: a version of `Index` highly optimized for 64-bit integer data, such as time stamps
- `Float64Index`: a version of `Index` highly optimized for 64-bit float data
- `MultiIndex`: the standard hierarchical index object
- `DatetimeIndex`: An `Index` object with `Timestamp` boxed elements (impl are the int64 values)
- `TimedeltaIndex`: An `Index` object with `Timedelta` boxed elements (impl are the in64 values)
- `PeriodIndex`: An `Index` object with `Period` elements

There are functions that make the creation of a regular index easy:

- `date_range`: fixed frequency date range generated from a time rule or `DateOffset`. An ndarray of Python datetime objects
- `period_range`: fixed frequency date range generated from a time rule or `DateOffset`. An ndarray of `Period` objects, representing Timespans

The motivation for having an `Index` class in the first place was to enable different implementations of indexing. This means that it's possible for you, the user, to implement a custom `Index` subclass that may be better suited to a particular application than the ones provided in pandas.

From an internal implementation point of view, the relevant methods that an `Index` must define are one or more of the following (depending on how incompatible the new object internals are with the `Index` functions):

- `get_loc`: returns an “indexer” (an integer, or in some cases a slice object) for a label
- `slice_locs`: returns the “range” to slice between two labels
- `get_indexer`: Computes the indexing vector for reindexing / data alignment purposes. See the source / docstrings for more on this
- `get_indexer_non_unique`: Computes the indexing vector for reindexing / data alignment purposes when the index is non-unique. See the source / docstrings for more on this
- `reindex`: Does any pre-conversion of the input index then calls `get_indexer`

- `union, intersection`: computes the union or intersection of two Index objects
- `insert`: Inserts a new label into an Index, yielding a new object
- `delete`: Delete a label, yielding a new object
- `drop`: Deletes a set of labels
- `take`: Analogous to `ndarray.take`

### 36.1.1 MultiIndex

Internally, the `MultiIndex` consists of a few things: the **levels**, the integer **labels**, and the level **names**:

```
In [1]: index = pd.MultiIndex.from_product([range(3), ['one', 'two']], names=['first',
↳ 'second'])

In [2]: index
Out[2]:
MultiIndex(levels=[[0, 1, 2], ['one', 'two']],
            labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]],
            names=['first', 'second'])

In [3]: index.levels
↳FrozenList([[0, 1, 2], ['one', 'two']])

In [4]: index.labels
↳FrozenList([[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])

In [5]: index.names
↳FrozenList(['first', 'second'])
```

You can probably guess that the labels determine which unique element is identified with that location at each layer of the index. It's important to note that sortedness is determined **solely** from the integer labels and does not check (or care) whether the levels themselves are sorted. Fortunately, the constructors `from_tuples` and `from_arrays` ensure that this is true, but if you compute the levels and labels yourself, please be careful.

### 36.1.2 Values

Pandas extends NumPy's type system with custom types, like `Categorical` or `datetimes` with a `timezone`, so we have multiple notions of “values”. For 1-D containers (Index classes and `Series`) we have the following convention:

- `cls._ndarray_values` is *always* a NumPy `ndarray`. Ideally, `_ndarray_values` is cheap to compute. For example, for a `Categorical`, this returns the codes, not the array of objects.
- `cls._values` refers to the “best possible” array. This could be an `ndarray`, `ExtensionArray`, or in Index subclass (note: we're in the process of removing the index subclasses here so that it's always an `ndarray` or `ExtensionArray`).

So, for example, `Series[category]._values` is a `Categorical`, while `Series[category]._ndarray_values` is the underlying codes.

## 36.2 Subclassing pandas Data Structures

This section has been moved to *Subclassing pandas Data Structures*.



## EXTENDING PANDAS

While pandas provides a rich set of methods, containers, and data types, your needs may not be fully satisfied. Pandas offers a few options for extending pandas.

### 37.1 Registering Custom Accessors

Libraries can use the decorators `pandas.api.extensions.register_dataframe_accessor()`, `pandas.api.extensions.register_series_accessor()`, and `pandas.api.extensions.register_index_accessor()`, to add additional “namespaces” to pandas objects. All of these follow a similar convention: you decorate a class, providing the name of attribute to add. The class’s `__init__` method gets the object being decorated. For example:

```
@pd.api.extensions.register_dataframe_accessor("geo")
class GeoAccessor(object):
    def __init__(self, pandas_obj):
        self._obj = pandas_obj

    @property
    def center(self):
        # return the geographic center point of this DataFrame
        lat = self._obj.latitude
        lon = self._obj.longitude
        return (float(lon.mean()), float(lat.mean()))

    def plot(self):
        # plot this array's data on a map, e.g., using Cartopy
        pass
```

Now users can access your methods using the `geo` namespace:

```
>>> ds = pd.DataFrame({'longitude': np.linspace(0, 10),
...                    'latitude': np.linspace(0, 20)})
>>> ds.geo.center
(5.0, 10.0)
>>> ds.geo.plot()
# plots data on a map
```

This can be a convenient way to extend pandas objects without subclassing them. If you write a custom accessor, make a pull request adding it to our [pandas Ecosystem](#) page.

## 37.2 Extension Types

New in version 0.23.0.

**Warning:** The `pandas.api.extension.ExtensionDtype` and `pandas.api.extension.ExtensionArray` APIs are new and experimental. They may change between versions without warning.

Pandas defines an interface for implementing data types and arrays that *extend* NumPy’s type system. Pandas itself uses the extension system for some types that aren’t built into NumPy (categorical, period, interval, datetime with timezone).

Libraries can define a custom array and data type. When pandas encounters these objects, they will be handled properly (i.e. not converted to an ndarray of objects). Many methods like `pandas.isna()` will dispatch to the extension type’s implementation.

If you’re building a library that implements the interface, please publicize it on [Extension Data Types](#).

The interface consists of two classes.

### 37.2.1 ExtensionDtype

A `pandas.api.extension.ExtensionDtype` is similar to a `numpy.dtype` object. It describes the data type. Implementors are responsible for a few unique items like the name.

One particularly important item is the `type` property. This should be the class that is the scalar type for your data. For example, if you were writing an extension array for IP Address data, this might be `ipaddress.IPv4Address`.

See the [extension dtype source](#) for interface definition.

### 37.2.2 ExtensionArray

This class provides all the array-like functionality. ExtensionArrays are limited to 1 dimension. An ExtensionArray is linked to an ExtensionDtype via the `dtype` attribute.

Pandas makes no restrictions on how an extension array is created via its `__new__` or `__init__`, and puts no restrictions on how you store your data. We do require that your array be convertible to a NumPy array, even if this is relatively expensive (as it is for `Categorical`).

They may be backed by none, one, or many NumPy arrays. For example, `pandas.Categorical` is an extension array backed by two arrays, one for codes and one for categories. An array of IPv6 addresses may be backed by a NumPy structured array with two fields, one for the lower 64 bits and one for the upper 64 bits. Or they may be backed by some other storage type, like Python lists.

See the [extension array source](#) for the interface definition. The docstrings and comments contain guidance for properly implementing the interface.

We provide a test suite for ensuring that your extension arrays satisfy the expected behavior. To use the test suite, you must provide several pytest fixtures and inherit from the base test class. The required fixtures are found in <https://github.com/pandas-dev/pandas/blob/master/pandas/tests/extension/conftest.py>.

To use a test, subclass it:

```
from pandas.tests.extension import base
```

(continues on next page)



(continued from previous page)

```
class TestConstructors(base.BaseConstructorsTests):
    pass
```

See [https://github.com/pandas-dev/pandas/blob/master/pandas/tests/extension/base/\\_\\_init\\_\\_.py](https://github.com/pandas-dev/pandas/blob/master/pandas/tests/extension/base/__init__.py) for a list of all the tests available.

## 37.3 Subclassing pandas Data Structures

**Warning:** There are some easier alternatives before considering subclassing pandas data structures.

1. Extensible method chains with *pipe*
2. Use *composition*. See [here](#).
3. Extending by *registering an accessor*
4. Extending by *extension type*

This section describes how to subclass pandas data structures to meet more specific needs. There are two points that need attention:

1. Override constructor properties.
2. Define original properties

**Note:** You can find a nice example in [geopandas](#) project.

### 37.3.1 Override Constructor Properties

Each data structure has several *constructor properties* for returning a new data structure as the result of an operation. By overriding these properties, you can retain subclasses through pandas data manipulations.

There are 3 constructor properties to be defined:

- `_constructor`: Used when a manipulation result has the same dimensions as the original.
- `_constructor_sliced`: Used when a manipulation result has one lower dimension(s) as the original, such as `DataFrame` single columns slicing.
- `_constructor_expanddim`: Used when a manipulation result has one higher dimension as the original, such as `Series.to_frame()` and `DataFrame.to_panel()`.

Following table shows how pandas data structures define constructor properties by default.

Property Attributes	Series	DataFrame
<code>_constructor</code>	Series	DataFrame
<code>_constructor_sliced</code>	NotImplementedError	Series
<code>_constructor_expanddim</code>	DataFrame	Panel

Below example shows how to define `SubclassedSeries` and `SubclassedDataFrame` overriding constructor properties.

```

class SubclassedSeries(Series):

    @property
    def _constructor(self):
        return SubclassedSeries

    @property
    def _constructor_expanddim(self):
        return SubclassedDataFrame

class SubclassedDataFrame(DataFrame):

    @property
    def _constructor(self):
        return SubclassedDataFrame

    @property
    def _constructor_sliced(self):
        return SubclassedSeries

```

```

>>> s = SubclassedSeries([1, 2, 3])
>>> type(s)
<class '__main__.SubclassedSeries'>

>>> to_framed = s.to_frame()
>>> type(to_framed)
<class '__main__.SubclassedDataFrame'>

>>> df = SubclassedDataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
>>> df
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9

>>> type(df)
<class '__main__.SubclassedDataFrame'>

>>> sliced1 = df[['A', 'B']]
>>> sliced1
   A  B
0  1  4
1  2  5
2  3  6
>>> type(sliced1)
<class '__main__.SubclassedDataFrame'>

>>> sliced2 = df['A']
>>> sliced2
0    1
1    2
2    3
Name: A, dtype: int64
>>> type(sliced2)
<class '__main__.SubclassedSeries'>

```

### 37.3.2 Define Original Properties

To let original data structures have additional properties, you should let pandas know what properties are added. pandas maps unknown properties to data names overriding `__getattr__`. Defining original properties can be done in one of 2 ways:

1. Define `_internal_names` and `_internal_names_set` for temporary properties which WILL NOT be passed to manipulation results.
2. Define `_metadata` for normal properties which will be passed to manipulation results.

Below is an example to define two original properties, “internal\_cache” as a temporary property and “added\_property” as a normal property

```
class SubclassedDataFrame2(DataFrame):

    # temporary properties
    _internal_names = pd.DataFrame._internal_names + ['internal_cache']
    _internal_names_set = set(_internal_names)

    # normal properties
    _metadata = ['added_property']

    @property
    def _constructor(self):
        return SubclassedDataFrame2
```

```
>>> df = SubclassedDataFrame2({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
>>> df
   A  B  C
0  1  4  7
1  2  5  8
2  3  6  9

>>> df.internal_cache = 'cached'
>>> df.added_property = 'property'

>>> df.internal_cache
cached
>>> df.added_property
property

# properties defined in _internal_names is reset after manipulation
>>> df[['A', 'B']].internal_cache
AttributeError: 'SubclassedDataFrame2' object has no attribute 'internal_cache'

# properties defined in _metadata are retained
>>> df[['A', 'B']].added_property
property
```



## RELEASE NOTES

This is the list of changes to pandas between each release. For full details, see the commit logs at <http://github.com/pandas-dev/pandas>

### What is it

pandas is a Python package providing fast, flexible, and expressive data structures designed to make working with “relational” or “labeled” data both easy and intuitive. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language.

### Where to get it

- Source code: <http://github.com/pandas-dev/pandas>
- Binary installers on PyPI: <https://pypi.org/project/pandas>
- Documentation: <http://pandas.pydata.org>

## 38.1 pandas 0.23.2

**Release date:** July 5, 2018

This is a minor bug-fix release in the 0.23.x series and includes some small regression fixes and bug fixes.

See the *full [whatsnew](#)* for a list of all the changes.

### 38.1.1 Thanks

A total of 17 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- David Krych
- Jacopo Rota +
- Jeff Reback
- Jeremy Schendel
- Joris Van den Bossche
- Kalyan Gokhale
- Matthew Roeschke
- Michael Odintsov +
- Ming Li

- Pietro Battiston
- Tom Augspurger
- Uddeshya Singh
- Vu Le +
- alimcmaster1 +
- david-liu-brattle-1 +
- gfyong
- jbrockmendel

## 38.2 pandas 0.23.1

**Release date:** June 12, 2018

This is a minor release from 0.23.0 and includes a number of bug fixes and performance improvements.

See the [full \*whatsnew\*](#) for a list of all the changes.

### 38.2.1 Thanks

A total of 30 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Adam J. Stewart
- Adam Kim +
- Aly Sivji
- Chalmer Lowe +
- Damini Satya +
- Dr. Irv
- Gabe Fernando +
- Giftlin Rajaiah
- Jeff Reback
- Jeremy Schendel +
- Joris Van den Bossche
- Kalyan Gokhale +
- Kevin Sheppard
- Matthew Roeschke
- Max Kanter +
- Ming Li
- Pyry Kovanen +
- Stefano Cianiulli
- Tom Augspurger

- Uddeshya Singh +
- Wenhuan
- William Ayd
- chris-b1
- gfyoun
- h-vetinari
- nprad +
- ssikdar1 +
- tmnh2001
- topper-123
- zertrin +

## 38.3 pandas 0.23.0

**Release date:** May 15, 2018

This is a major release from 0.22.0 and includes a number of API changes, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- *Round-trippable JSON format with ‘table’ orient.*
- *Instantiation from dicts respects order for Python 3.6+.*
- *Dependent column arguments for assign.*
- *Merging / sorting on a combination of columns and index levels.*
- *Extending Pandas with custom types.*
- *Excluding unobserved categories from groupby.*
- *Changes to make output shape of DataFrame.apply consistent.*

See the *full whatsnew* for a list of all the changes.

### 38.3.1 Thanks

A total of 328 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

- Aaron Critchley
- AbdealJK +
- Adam Hooper +
- Albert Villanova del Moral
- Alejandro Giacometti +
- Alejandro Hohmann +
- Alex Rychyk

- Alexander Buchkovsky
- Alexander Lenail +
- Alexander Michael Schade
- Aly Sivji +
- Andreas Költringer +
- Andrew
- Andrew Bui +
- András Novoszáth +
- Andy Craze +
- Andy R. Terrel
- Anh Le +
- Anil Kumar Pallekonda +
- Antoine Pitrou +
- Antonio Linde +
- Antonio Molina +
- Antonio Quinonez +
- Armin Varshokar +
- Artem Bogachev +
- Avi Sen +
- Azeez Oluwafemi +
- Ben Auffarth +
- Bernhard Thiel +
- Bhavesh Poddar +
- BielStela +
- Blair +
- Bob Haffner
- Brett Naul +
- Brock Mendel
- Bryce Guinta +
- Carlos Eduardo Moreira dos Santos +
- Carlos García Márquez +
- Carol Willing
- Cheuk Ting Ho +
- Chitrang Dixit +
- Chris
- Chris Burr +



- Chris Catalfo +
- Chris Mazzullo
- Christian Chwala +
- Cihan Ceyhan +
- Clemens Brunner
- Colin +
- Cornelius Riemenschneider
- Crystal Gong +
- DaanVanHauwermeiren
- Dan Dixey +
- Daniel Frank +
- Daniel Garrido +
- Daniel Sakuma +
- DataOmbudsman +
- Dave Hirschfeld
- Dave Lewis +
- David Adrián Cañones Castellano +
- David Arcos +
- David C Hall +
- David Fischer
- David Hoese +
- David Lutz +
- David Polo +
- David Stansby
- Dennis Kamau +
- Dillon Niederhut
- Dimitri +
- Dr. Irv
- Dror Atariah
- Eric Chea +
- Eric Kisslinger
- Eric O. LEBIGOT (EOL) +
- FAN-GOD +
- Fabian Retkowski +
- Fer Sar +
- Gabriel de Maeztu +

- Gianpaolo Macario +
- Giftlin Rajaiah
- Gilberto Olimpio +
- Gina +
- Gjelt +
- Graham Inggs +
- Grant Roch
- Grant Smith +
- Grzegorz Konefał +
- Guilherme Beltramini
- HagaiHargil +
- Hamish Pitkeathly +
- Hammad Mashkooor +
- Hannah Ferchland +
- Hans
- Haochen Wu +
- Hissashi Rocha +
- Iain Barr +
- Ibrahim Sharaf ElDen +
- Ignasi Fosch +
- Igor Conrado Alves de Lima +
- Igor Shelvinskyi +
- Imanflow +
- Ingolf Becker
- Israel Saeta Pérez
- Iva Koevska +
- Jakub Nowacki +
- Jan F-F +
- Jan Koch +
- Jan Werkmann
- Janelle Zoutkamp +
- Jason Bandlow +
- Jaume Bonet +
- Jay Alammam +
- Jeff Reback
- JennaVergeynst

- Jimmy Woo +
- Jing Qiang Goh +
- Joachim Wagner +
- Joan Martin Miralles +
- Joel Nothman
- Joeun Park +
- John Cant +
- Johnny Metz +
- Jon Mease
- Jonas Schulze +
- Jongwony +
- Jordi Contestí +
- Joris Van den Bossche
- José F. R. Fonseca +
- Jovixe +
- Julio Martinez +
- Jörg Döpfert
- KOBAYASHI Ittoku +
- Kate Surta +
- Kenneth +
- Kevin Kuhl
- Kevin Sheppard
- Krzysztof Chomski
- Ksenia +
- Ksenia Bobrova +
- Kunal Gosar +
- Kurtis Kerstein +
- Kyle Barron +
- Laksh Arora +
- Laurens Geffert +
- Leif Walsh
- Liam Marshall +
- Liam3851 +
- Licht Takeuchi
- Liudmila +
- Ludovico Russo +

- Mabel Villalba +
- Manan Pal Singh +
- Manraj Singh
- Marc +
- Marc Garcia
- Marco Hemken +
- Maria del Mar Bibiloni +
- Mario Corchero +
- Mark Woodbridge +
- Martin Journois +
- Mason Gallo +
- Matias Heikkilä +
- Matt Braymer-Hayes
- Matt Kirk +
- Matt Maybeno +
- Matthew Kirk +
- Matthew Rocklin +
- Matthew Roeschke
- Matthias Bussonnier +
- Max Mikhaylov +
- Maxim Veksler +
- Maximilian Roos
- Maximiliano Greco +
- Michael Penkov
- Michael Röttger +
- Michael Selik +
- Michael Waskom
- Mie~~~
- Mike Kutzma +
- Ming Li +
- Mitar +
- Mitch Negus +
- Montana Low +
- Moritz Müntz +
- Mortada Mehyar
- Myles Braithwaite +

- Nate Yoder
- Nicholas Ursa +
- Nick Chmura
- Nikos Karagiannakis +
- Nipun Sadvilkar +
- Nis Martensen +
- Noah +
- Noémi Éltető +
- Olivier Bilodeau +
- Ondrej Kokes +
- Onno Eberhard +
- Paul Ganssle +
- Paul Mannino +
- Paul Reidy
- Paulo Roberto de Oliveira Castro +
- Pepe Flores +
- Peter Hoffmann
- Phil Ngo +
- Pietro Battiston
- Pranav Suri +
- Priyanka Ojha +
- Pulkit Maloo +
- README Bot +
- Ray Bell +
- Riccardo Magliocchetti +
- Ridhwan Luthra +
- Robert Meyer
- Robin
- Robin Kiplang'at +
- Rohan Pandit +
- Rok Mihevc +
- Rouz Azari
- Ryszard T. Kaleta +
- Sam Cohan
- Sam Foo
- Samir Musali +

- Samuel Sinayoko +
- Sangwoong Yoon
- SarahJessica +
- Sharad Vijalapuram +
- Shubham Chaudhary +
- SiYoungOh +
- Sietse Brouwer
- Simone Basso +
- Stefania Delprete +
- Stefano Cianciulli +
- Stephen Childs +
- StephenVoland +
- Stijn Van Hoey +
- Sven
- Talitha Pumar +
- Tarbo Fukazawa +
- Ted Petrou +
- Thomas A Caswell
- Tim Hoffmann +
- Tim Swast
- Tom Augspurger
- Tommy +
- Tulio Casagrande +
- Tushar Gupta +
- Tushar Mittal +
- Upkar Lidder +
- Victor Villas +
- Vince W +
- Vinícius Figueiredo +
- Vipin Kumar +
- WBare
- Wenhuan +
- Wes Turner
- William Ayd
- Wilson Lin +
- Xbar

- Yaroslav Halchenko
- Yee Mey
- Yeongseon Choe +
- Yian +
- Yimeng Zhang
- ZhuBaohe +
- Zihao Zhao +
- adatasetaday +
- akielbowicz +
- akosel +
- alinde1 +
- amuta +
- bolkedebruin
- cbertinato
- cgohlke
- charlie0389 +
- chris-b1
- csfarkas +
- dajcs +
- deflatSOCO +
- derestle-htwg
- discort
- dmanikowski-reef +
- donK23 +
- elrubio +
- fivemok +
- fjdioid
- fjetter +
- froessler +
- gabrielclow
- gfyong
- ghasemnaddaf
- h-vetinari +
- himanshu awasthi +
- ignamv +
- jayfoad +

- jazzmuesli +
- jbrockmendel
- jen w +
- jjames34 +
- joaoavf +
- joders +
- jschendel
- juan huguet +
- l736x +
- luzpaz +
- mdeboc +
- miguelmorin +
- miker985
- miquelcamprodon +
- orereta +
- ottiP +
- peterpanmj +
- rafarui +
- raph-m +
- readyready15728 +
- rmihael +
- samghelms +
- scriptomation +
- sfoo +
- stefansimik +
- stonebig
- tmnhathat2001 +
- tomneep +
- topper-123
- tv3141 +
- verakai +
- xpvpc +
- zhanghui +



## 38.4 pandas 0.22.0

**Release date:** December 29, 2017

This is a major release from 0.21.1 and includes a single, API-breaking change. We recommend that all users upgrade to this version after carefully reading the release note.

The only changes are:

- The sum of an empty or all-NA Series is now 0
- The product of an empty or all-NA Series is now 1
- We've added a `min_count` parameter to `.sum()` and `.prod()` controlling the minimum number of valid values for the result to be valid. If fewer than `min_count` non-NA values are present, the result is NA. The default is 0. To return NaN, the 0.21 behavior, use `min_count=1`.

See the [v0.22.0 Whatsnew](#) overview for further explanation of all the places in the library this affects.

## 38.5 pandas 0.21.1

**Release date:** December 12, 2017

This is a minor bug-fix release in the 0.21.x series and includes some small regression fixes, bug fixes and performance improvements. We recommend that all users upgrade to this version.

Highlights include:

- Temporarily restore matplotlib datetime plotting functionality. This should resolve issues for users who relied implicitly on pandas to plot datetimes with matplotlib. See [here](#).
- Improvements to the Parquet IO functions introduced in 0.21.0. See [here](#).

See the [v0.21.1 Whatsnew](#) overview for an extensive list of all the changes for 0.21.1.

### 38.5.1 Thanks

A total of 46 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

#### 38.5.1.1 Contributors

- Aaron Critchley +
- Alex Rychyk
- Alexander Buchkovsky +
- Alexander Michael Schade +
- Chris Mazzullo
- Cornelius Riemenschneider +
- Dave Hirschfeld +
- David Fischer +
- David Stansby +
- Dror Atariah +

- Eric Kisslinger +
- Hans +
- Ingolf Becker +
- Jan Werkmann +
- Jeff Reback
- Joris Van den Bossche
- Jörg Döpfert +
- Kevin Kuhl +
- Krzysztof Chomski +
- Leif Walsh
- Licht Takeuchi
- Manraj Singh +
- Matt Braymer-Hayes +
- Michael Waskom +
- Mie~~~ +
- Peter Hoffmann +
- Robert Meyer +
- Sam Cohan +
- Sietse Brouwer +
- Sven +
- Tim Swast
- Tom Augspurger
- Wes Turner
- William Ayd +
- Yee Mey +
- bolkedebruin +
- cgohlke
- derestle-htwg +
- fjdioid +
- gabrielclow +
- gfyong
- ghasemnaddaf +
- jbrockmendel
- jschendel
- miker985 +
- topper-123

## 38.6 pandas 0.21.0

**Release date:** October 27, 2017

This is a major release from 0.20.3 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Integration with [Apache Parquet](#), including a new top-level `read_parquet()` function and `DataFrame.to_parquet()` method, see [here](#).
- New user-facing `pandas.api.types.CategoricalDtype` for specifying categoricals independent of the data, see [here](#).
- The behavior of `sum` and `prod` on all-NaN Series/DataFrames is now consistent and no longer depends on whether `bottleneck` is installed, and `sum` and `prod` on empty Series now return NaN instead of 0, see [here](#).
- Compatibility fixes for pypy, see [here](#).
- Additions to the `drop`, `reindex` and `rename` API to make them more consistent, see [here](#).
- Addition of the new methods `DataFrame.infer_objects` (see [here](#)) and `GroupBy.pipe` (see [here](#)).
- Indexing with a list of labels, where one or more of the labels is missing, is deprecated and will raise a `KeyError` in a future version, see [here](#).

See the [v0.21.0 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.21.0

### 38.6.1 Thanks

A total of 206 people contributed to this release. People with a “+” by their names contributed a patch for the first time.

#### 38.6.1.1 Contributors

- 3553x +
- Aaron Barber
- Adam Gleave +
- Adam Smith +
- AdamShamlan +
- Adrian Liaw +
- Alan Velasco +
- Alan Yee +
- Alex B +
- Alex Lubbock +
- Alex Marchenko +
- Alex Rychyk +
- Amol K +

- Andreas Winkler
- Andrew +
- Andrew
- André Jonasson +
- Becky Sweger
- Berkay +
- Bob Haffner +
- Bran Yang
- Brian Tu +
- Brock Mendel +
- Carol Willing +
- Carter Green +
- Chankey Pathak +
- Chris
- Chris Billington
- Chris Filo Gorgolewski +
- Chris Kerr
- Chris M +
- Chris Mazzullo +
- Christian Prinoth
- Christian Stade-Schuldt
- Christoph Moehl +
- DSM
- Daniel Chen +
- Daniel Grady
- Daniel Himmelstein
- Dave Willmer
- David Cook
- David Gwynne
- David Read +
- Dillon Niederhut +
- Douglas Rudd
- Eric Stein +
- Eric Wieser +
- Erik Fredriksen
- Florian Wilhelm +

- Floris Kint +
- Forbidden Donut
- Gabe F +
- Giftlin +
- Giftlin Rajaiah +
- Giulio Pepe +
- Guilherme Beltramini
- Guillem Borrell +
- Hanmin Qin +
- Hendrik Makait +
- Hugues Valois
- Hussain Tamboli +
- Iva Miholic +
- Jan Novotný +
- Jan Rudolph
- Jean Helie +
- Jean-Baptiste Schiratti +
- Jean-Mathieu Deschenes
- Jeff Knupp +
- Jeff Reback
- Jeff Tratner
- JennaVergeynst
- JimStearns206
- Joel Nothman
- John W. O'Brien
- Jon Crall +
- Jon Mease
- Jonathan J. Helmus +
- Joris Van den Bossche
- JosephWagner
- Juarez Bochi
- Julian Kuhlmann +
- Karel De Brabandere
- Kassandra Keeton +
- Keiron Pizzey +
- Keith Webber

- Kernc
- Kevin Sheppard
- Kirk Hansen +
- Licht Takeuchi +
- Lucas Kushner +
- Mahdi Ben Jelloul +
- Makarov Andrey +
- Malgorzata Turzanska +
- Marc Garcia +
- Margaret Sy +
- MarsGuy +
- Matt Bark +
- Matthew Roeschke
- Matti Picus
- Mehmet Ali “Mali” Akmanalp
- Michael Gasvoda +
- Michael Penkov +
- Milo +
- Morgan Stuart +
- Morgan243 +
- Nathan Ford +
- Nick Eubank
- Nick Garvey +
- Oleg Shteynbuk +
- P-Tillmann +
- Pankaj Pandey
- Patrick Luo
- Patrick O’Melveny
- Paul Reidy +
- Paula +
- Peter Quackenbush
- Peter Yanovich +
- Phillip Cloud
- Pierre Haessig
- Pietro Battiston
- Pradyumna Reddy Chinthala

- Prasanjit Prakash
- RobinFiveWords
- Ryan Hendrickson
- Sam Foo
- Sangwoong Yoon +
- Simon Gibbons +
- SimonBaron
- Steven Cutting +
- Sudeep +
- Sylvia +
- T N +
- Telt
- Thomas A Caswell
- Tim Swast +
- Tom Augspurger
- Tong SHEN
- Tuan +
- Utkarsh Upadhyay +
- Vincent La +
- Vivek +
- WANG Aiyong
- WBare
- Wes McKinney
- XF +
- Yi Liu +
- Yosuke Nakabayashi +
- aaron315 +
- abarber4gh +
- aernlund +
- agustín méndez +
- andymaheshw +
- ante328 +
- aviolov +
- bpraggastis
- cbertinato +
- cclauss +

- [chernrick](#)
- [chris-b1](#)
- [dkamm](#) +
- [dwkenefick](#)
- [economy](#)
- [faic](#) +
- [fding253](#) +
- [gfyong](#)
- [guygoldberg](#) +
- [hhuuggoo](#) +
- [huashuai](#) +
- [ian](#)
- [iulia](#) +
- [jaredsnyder](#)
- [jbrockmendel](#) +
- [jdeschenes](#)
- [jebob](#) +
- [jschendel](#) +
- [keitakurita](#)
- [kernc](#) +
- [kiwirob](#) +
- [kjford](#)
- [linebp](#)
- [lloydkirk](#)
- [louispotok](#) +
- [majiang](#) +
- [manikbhandari](#) +
- [matthiashuschle](#) +
- [mattip](#)
- [maxwasserman](#) +
- [mjlove12](#) +
- [nmartensen](#) +
- [pandas-docs-bot](#) +
- [parchd-1](#) +
- [philipphanemann](#) +
- [rdk1024](#) +



- reidy-p +
- ri938
- ruiann +
- rvernica +
- s-weigand +
- scotthavard92 +
- skwbc +
- step4me +
- tobycheese +
- toppe-123 +
- tsdlovell
- ysau +
- zzgao +

## 38.7 pandas 0.20.0 / 0.20.1

**Release date:** May 5, 2017

This is a major release from 0.19.2 and includes a number of API changes, deprecations, new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- New `.agg()` API for Series/DataFrame similar to the groupby-rolling-resample API's, see [here](#)
- Integration with the feather-format, including a new top-level `pd.read_feather()` and `DataFrame.to_feather()` method, see [here](#).
- The `.ix` indexer has been deprecated, see [here](#)
- `Panel` has been deprecated, see [here](#)
- Addition of an `IntervalIndex` and `Interval` scalar type, see [here](#)
- Improved user API when grouping by index levels in `.groupby()`, see [here](#)
- Improved support for `UInt64` dtypes, see [here](#)
- A new orient for JSON serialization, `orient='table'`, that uses the Table Schema spec and that gives the possibility for a more interactive repr in the Jupyter Notebook, see [here](#)
- Experimental support for exporting styled DataFrames (`DataFrame.style`) to Excel, see [here](#)
- Window binary `corr/cov` operations now return a `MultiIndexed DataFrame` rather than a `Panel`, as `Panel` is now deprecated, see [here](#)
- Support for S3 handling now uses `s3fs`, see [here](#)
- Google BigQuery support now uses the `pandas-gbq` library, see [here](#)

See the [v0.20.1 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.20.1.

---

**Note:** This is a combined release for 0.20.0 and 0.20.1. Version 0.20.1 contains one additional change for backwards-compatibility with downstream projects using pandas' `utils` routines. ([GH16250](#))

---

### 38.7.1 Thanks

- [abaldenko](#)
- [Adam J. Stewart](#)
- [Adrian](#)
- [adrian-stepien](#)
- [Ajay Saxena](#)
- [Akash Tandon](#)
- [Albert Villanova del Moral](#)
- [Aleksey Bilogur](#)
- [alexandercbooth](#)
- [Alexis Mignon](#)
- [Amol Kahat](#)
- [Andreas Winkler](#)
- [Andrew Kittredge](#)
- [Anthonios Partheniou](#)
- [Arco Bast](#)
- [Ashish Singal](#)
- [atbd](#)
- [bastewart](#)
- [Baurzhan Muftakhidinov](#)
- [Ben Kandel](#)
- [Ben Thayer](#)
- [Ben Welsh](#)
- [Bill Chambers](#)
- [bmagnusson](#)
- [Brandon M. Burroughs](#)
- [Brian](#)
- [Brian McFee](#)
- [carlosdanielcsantos](#)
- [Carlos Souza](#)
- [chaimdemulder](#)

- Chris
- chris-b1
- Chris Ham
- Christopher C. Aycock
- Christoph Gohlke
- Christoph Paulik
- Chris Warth
- Clemens Brunner
- DaanVanHauwermeiren
- Daniel Himmelstein
- Dave Willmer
- David Cook
- David Gwynne
- David Hoffman
- David Krych
- dickreuter
- Diego Fernandez
- Dimitris Spathis
- discort
- Dmitry L
- Dody Suria Wijaya
- Dominik Stanczak
- Dr-Irv
- Dr. Irv
- dr-leo
- D.S. McNeil
- dubourg
- dwkenefick
- Elliott Sales de Andrade
- Ennemoser Christoph
- Francesc Alted
- Fumito Hamamura
- funnycrab
- gfyoun
- Giacomo Ferroni
- goldenbull

- Graham R. Jeffries
- Greg Williams
- Guilherme Beltrami
- Guilherme Samora
- Hao Wu
- Harshit Patni
- [hesham.shabana@hotmail.com](mailto:hesham.shabana@hotmail.com)
- Ilya V. Schurov
- Iván Vallés Pérez
- Jackie Leng
- Jaehoon Hwang
- James Draper
- James Goppert
- James McBride
- James Santucci
- Jan Schulz
- Jeff Carey
- Jeff Reback
- JennaVergeynst
- Jim
- Jim Crist
- Joe Jevnik
- Joel Nothman
- John
- John Tucker
- John W. O'Brien
- John Zwinck
- jojomdt
- Jonathan de Bruin
- Jonathan Whitmore
- Jon Mease
- Jon M. Mease
- Joost Kranendonk
- Joris Van den Bossche
- Joshua Bradt
- Julian Santander

- Julien Marrec
- Jun Kim
- Justin Solinsky
- Kacawi
- Kamal Kamalaldin
- Kerby Shedden
- Kernc
- Keshav Ramaswamy
- Kevin Sheppard
- Kyle Kelley
- Larry Ren
- Leon Yin
- linebp
- Line Pedersen
- Lorenzo Cestaro
- Luca Scarabello
- Lukasz
- Mahmoud Lababidi
- manu
- manuels
- Mark Mandel
- Matthew Brett
- Matthew Roeschke
- mattip
- Matti Picus
- Matt Roeschke
- maxalbert
- Maximilian Roos
- mcocdawc
- Michael Charlton
- Michael Felt
- Michael Lamparski
- Michiel Stock
- Mikolaj Chwalisz
- Min RK
- Miroslav Šedivý

- Mykola Golubyev
- Nate Yoder
- Nathalie Rud
- Nicholas Ver Halen
- Nick Chmura
- Nolan Nichols
- nuffe
- Pankaj Pandey
- paul-mannino
- Pawel Kordek
- pbreach
- Pete Huang
- Peter
- Peter Csizsek
- Petio Petrov
- Phil Ruffwind
- Pietro Battiston
- Piotr Chromiec
- Prasanjit Prakash
- Robert Bradshaw
- Rob Forgione
- Robin
- Rodolfo Fernandez
- Roger Thomas
- Rouz Azari
- Sahil Dua
- sakkemo
- Sam Foo
- Sami Salonen
- Sarah Bird
- Sarma Tangirala
- scls19fr
- Scott Sanderson
- Sebastian Bank
- Sebastian Gsänger
- Sébastien de Menten

- Shawn Heide
- Shyam Saladi
- sinhrks
- Sinhrks
- Stephen Rauch
- stijnavhoeey
- Tara Adiseshan
- themrmax
- the-nose-knows
- Thiago Serafim
- Thoralf Gutierrez
- Thrasibule
- Tobias Gustafsson
- Tom Augspurger
- tomrod
- Tong Shen
- Tong SHEN
- TrigonaMinima
- tzinckgraf
- Uwe
- wandersoncferreira
- watercrossing
- wcwagner
- Wes Turner
- Wiktoria Tomczak
- WillAyd
- xgdgsc
- Yaroslav Halchenko
- Yimeng Zhang
- yui-knk

## 38.8 pandas 0.19.2

**Release date:** December 24, 2016

This is a minor bug-fix release in the 0.19.x series and includes some small regression fixes, bug fixes and performance improvements.

Highlights include:

- Compatibility with Python 3.6
- Added a [Pandas Cheat Sheet](#). (GH13202).

See the [v0.19.2 Whatsnew](#) page for an overview of all bugs that have been fixed in 0.19.2.

### 38.8.1 Thanks

- Ajay Saxena
- Ben Kandel
- Chris
- Chris Ham
- Christopher C. Aycock
- Daniel Himmelstein
- Dave Willmer
- Dr-Irv
- gfyong
- hesham shabana
- Jeff Carey
- Jeff Reback
- Joe Jevnik
- Joris Van den Bossche
- Julian Santander
- Kerby Shedden
- Keshav Ramaswamy
- Kevin Sheppard
- Luca Scarabello
- Matti Picus
- Matt Roeschke
- Maximilian Roos
- Mykola Golubyev
- Nate Yoder
- Nicholas Ver Halen
- Pawel Kordek
- Pietro Battiston
- Rodolfo Fernandez
- sinhrks
- Tara Adiseshan
- Tom Augspurger



- wandersoncferreira
- Yaroslav Halchenko

## 38.9 pandas 0.19.1

**Release date:** November 3, 2016

This is a minor bug-fix release from 0.19.0 and includes some small regression fixes, bug fixes and performance improvements.

See the [v0.19.1 Whatsnew](#) page for an overview of all bugs that have been fixed in 0.19.1.

### 38.9.1 Thanks

- Adam Chainz
- Anthonios Partheniou
- Arash Rouhani
- Ben Kandel
- Brandon M. Burroughs
- Chris
- chris-b1
- Chris Warth
- David Krych
- dubourg
- gfyoun
- Iván Vallés Pérez
- Jeff Reback
- Joe Jevnik
- Jon M. Mease
- Joris Van den Bossche
- Josh Owen
- Keshav Ramaswamy
- Larry Ren
- matrijk
- Michael Felt
- paul-mannino
- Piotr Chromiec
- Robert Bradshaw
- Sinhrks

- Thiago Serafim
- Tom Bird

## 38.10 pandas 0.19.0

**Release date:** October 2, 2016

This is a major release from 0.18.1 and includes number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- `merge_asof()` for asof-style time-series joining, see [here](#)
- `.rolling()` is now time-series aware, see [here](#)
- `read_csv()` now supports parsing Categorical data, see [here](#)
- A function `union_categorical()` has been added for combining categoricals, see [here](#)
- `PeriodIndex` now has its own `period` dtype, and changed to be more consistent with other `Index` classes. See [here](#)
- Sparse data structures gained enhanced support of `int` and `bool` dtypes, see [here](#)
- Comparison operations with `Series` no longer ignores the index, see [here](#) for an overview of the API changes.
- Introduction of a pandas development API for utility functions, see [here](#).
- Deprecation of `Panel4D` and `PanelND`. We recommend to represent these types of n-dimensional data with the `xarray` package.
- Removal of the previously deprecated modules `pandas.io.data`, `pandas.io.wb`, `pandas.tools.rplot`.

See the [v0.19.0 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.19.0.

### 38.10.1 Thanks

- adneu
- Adrien Emery
- agraboso
- Alex Alekseyev
- Alex Vig
- Allen Riddell
- Amol
- Amol Agrawal
- Andy R. Terrel
- Anthonios Partheniou
- babakkeyvani
- Ben Kandel

- Bob Baxley
- Brett Rosen
- c123w
- Camilo Cota
- Chris
- chris-b1
- Chris Grinolds
- Christian Hudon
- Christopher C. Aycock
- Chris Warth
- cmazzullo
- conquistador1492
- cr3
- Daniel Siladji
- Douglas McNeil
- Drewrey Lupton
- dsm054
- Eduardo Blancas Reyes
- Elliot Marsden
- Evan Wright
- Felix Marczinowski
- Francis T. O'Donovan
- Gábor Lipták
- Geraint Duck
- gfyong
- Giacomo Ferroni
- Grant Roch
- Haleemur Ali
- harshul1610
- Hassan Shamim
- iamsimha
- Iulius Curt
- Ivan Nazarov
- jackieleng
- Jeff Reback
- Jeffrey Gerard

- Jenn Olsen
- Jim Crist
- Joe Jevnik
- John Evans
- John Freeman
- John Liekezer
- Johnny Gill
- John W. O'Brien
- John Zwinck
- Jordan Erenrich
- Joris Van den Bossche
- Josh Howes
- Jozef Brandys
- Kamil Sindi
- Ka Wo Chen
- Kerby Shedden
- Kernc
- Kevin Sheppard
- Matthieu Brucher
- Maximilian Roos
- Michael Scherer
- Mike Graham
- Mortada Mehyar
- mpuels
- Muhammad Haseeb Tariq
- Nate George
- Neil Parley
- Nicolas Bonnotte
- OXPHOS
- Pan Deng / Zora
- Paul
- Pauli Virtanen
- Paul Mestemaker
- Pawel Kordek
- Pietro Battiston
- pijucha

- Piotr Jucha
- priyankjain
- Ravi Kumar Nimmi
- Robert Gieseke
- Robert Kern
- Roger Thomas
- Roy Keyes
- Russell Smith
- Sahil Dua
- Sanjiv Lobo
- Sašo Stanovnik
- Shawn Heide
- sinhrks
- Sinhrks
- Stephen Kappel
- Steve Choi
- Stewart Henderson
- Sudarshan Konge
- Thomas A Caswell
- Tom Augspurger
- Tom Bird
- Uwe Hoffmann
- wcwagner
- WillAyd
- Xiang Zhang
- Yadunandan
- Yaroslav Halchenko
- YG-Riku
- Yuichiro Kaneko
- yui-knk
- zhangjinjie
- znmean
- Yan Facai

## 38.11 pandas 0.18.1

**Release date:** (May 3, 2016)

This is a minor release from 0.18.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements.

Highlights include:

- `.groupby(...)` has been enhanced to provide convenient syntax when working with `.rolling(...)`, `.expanding(...)` and `.resample(...)` per group, see [here](#)
- `pd.to_datetime()` has gained the ability to assemble dates from a DataFrame, see [here](#)
- Method chaining improvements, see [here](#).
- Custom business hour offset, see [here](#).
- Many bug fixes in the handling of sparse, see [here](#)
- Expanded the *Tutorials section* with a feature on modern pandas, courtesy of @TomAugsburger. (GH13045).

See the [v0.18.1 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.18.1.

### 38.11.1 Thanks

- Andrew Fiore-Gartland
- Bastiaan
- Benoît Vinot
- Brandon Rhodes
- DaCoEx
- Drew Fustin
- Ernesto Freitas
- Filip Ter
- Gregory Livschitz
- Gábor Lipták
- Hassan Kibirige
- Iblis Lin
- Israel Saeta Pérez
- Jason Wolosonovich
- Jeff Reback
- Joe Jevnik
- Joris Van den Bossche
- Joshua Storck
- Ka Wo Chen
- Kerby Shedden
- Kieran O'Mahony

- Leif Walsh
- Mahmoud Lababidi
- Maoyuan Liu
- Mark Roth
- Matt Wittmann
- MaxU
- Maximilian Roos
- Michael Droettboom
- Nick Eubank
- Nicolas Bonnotte
- OXPHOS
- Pauli Virtanen
- Peter Waller
- Pietro Battiston
- Prabhjot Singh
- Robin Wilson
- Roger Thomas
- Sebastian Bank
- Stephen Hoover
- Tim Hopper
- Tom Augspurger
- WANG Aiyong
- Wes Turner
- Winand
- Xbar
- Yan Facai
- adneu
- ajenkins-cargometrics
- behzad nouri
- chinskiy
- gfyong
- jeps-journal
- jonaslb
- kotrfa
- nileracecrew
- onesandzeroes

- rs2
- sinhrks
- tsdlovell

## 38.12 pandas 0.18.0

**Release date:** (March 13, 2016)

This is a major release from 0.17.1 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Moving and expanding window functions are now methods on Series and DataFrame, similar to `.groupby`, see [here](#).
- Adding support for a `RangeIndex` as a specialized form of the `Int64Index` for memory savings, see [here](#).
- API breaking change to the `.resample` method to make it more `.groupby` like, see [here](#).
- Removal of support for positional indexing with floats, which was deprecated since 0.14.0. This will now raise a `TypeError`, see [here](#).
- The `.to_xarray()` function has been added for compatibility with the `xarray` package, see [here](#).
- The `read_sas` function has been enhanced to read `sas7bdat` files, see [here](#).
- Addition of the `.str.extractall()` method, and API changes to the `.str.extract()` method and `.str.cat()` method.
- `pd.test()` top-level nose test runner is available ([GH4327](#)).

See the [v0.18.0 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.18.0.

### 38.12.1 Thanks

- ARF
- Alex Alekseyev
- Andrew McPherson
- Andrew Rosenfeld
- Anthonios Partheniou
- Anton I. Sipos
- Ben
- Ben North
- Bran Yang
- Chris
- Chris Carroux
- Christopher C. Aycock
- Christopher Scanlin



- Cody
- Da Wang
- Daniel Grady
- Dorozhko Anton
- Dr-Irv
- Erik M. Bray
- Evan Wright
- Francis T. O'Donovan
- Frank Cleary
- Gianluca Rossi
- Graham Jeffries
- Guillaume Horel
- Henry Hammond
- Isaac Schwabacher
- Jean-Mathieu Deschenes
- Jeff Reback
- Joe Jevnik
- John Freeman
- John Fremlin
- Jonas Hoersch
- Joris Van den Bossche
- Joris Vankerschaver
- Justin Lecher
- Justin Lin
- Ka Wo Chen
- Keming Zhang
- Kerby Shedden
- Kyle
- Marco Farrugia
- MasonGallo
- MattRijk
- Matthew Lurie
- Maximilian Roos
- Mayank Asthana
- Mortada Mehyar
- Moussa Taifi

- Navreet Gill
- Nicolas Bonnotte
- Paul Reiners
- Philip Gura
- Pietro Battiston
- RahulHP
- Randy Carnevale
- Rinoc Johnson
- Rishipuri
- Sangmin Park
- Scott E Lasley
- Sereger13
- Shannon Wang
- Skipper Seabold
- Thierry Moisan
- Thomas A Caswell
- Toby Dylan Hocking
- Tom Augspurger
- Travis
- Trent Hauck
- Tux1
- Varun
- Wes McKinney
- Will Thompson
- Yoav Ram
- Yoong Kang Lim
- Yoshiki Vázquez Baeza
- Young Joong Kim
- Younggun Kim
- Yuval Langer
- alex argunov
- behzad nouri
- boombard
- brian-pantano
- chromy
- daniel

- dgram0
- gfyoun
- hack-c
- hcontrast
- jfoo
- kaustuv deolal
- llllllllll
- ranarag
- rockg
- scls19fr
- seales
- sinhrks
- srib
- surveymedia.ca
- tworec

## 38.13 pandas 0.17.1

**Release date:** (November 21, 2015)

This is a minor release from 0.17.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements.

Highlights include:

- Support for Conditional HTML Formatting, see [here](#)
- Releasing the GIL on the csv reader & other ops, see [here](#)
- Regression in `DataFrame.drop_duplicates` from 0.16.2, causing incorrect results on integer values ([GH11376](#))

See the [v0.17.1 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.17.1.

### 38.13.1 Thanks

- Aleksandr Drozd
- Alex Chase
- Anthonios Partheniou
- BrenBarn
- Brian J. McGuirk
- Chris
- Christian Berendt
- Christian Perez

- Cody Piersall
- Data & Code Expert Experimenting with Code on Data
- DrIrv
- Evan Wright
- Guillaume Gay
- Hamed Saljooghinejad
- Iblis Lin
- Jake VanderPlas
- Jan Schulz
- Jean-Mathieu Deschenes
- Jeff Reback
- Jimmy Callin
- Joris Van den Bossche
- K.-Michael Aye
- Ka Wo Chen
- Loïc Séguin-C
- Luo Yicheng
- Magnus Jöud
- Manuel Leonhardt
- Matthew Gilbert
- Maximilian Roos
- Michael
- Nicholas Stahl
- Nicolas Bonnotte
- Pastafarianist
- Petra Chong
- Phil Schaf
- Philipp A
- Rob deCarvalho
- Roman Khomenko
- Rémy Léone
- Sebastian Bank
- Thierry Moisan
- Tom Augspurger
- Tux l
- Varun

- Wieland Hoffmann
- Winterflower
- Yoav Ram
- Younggun Kim
- Zeke
- ajcr
- azuranski
- behzad nouri
- cel4
- emilydolson
- hironow
- lexical
- llllllllll
- rockg
- silentquasar
- sinhrks
- taeold

## 38.14 pandas 0.17.0

**Release date:** (October 9, 2015)

This is a major release from 0.16.2 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes. We recommend that all users upgrade to this version.

Highlights include:

- Release the Global Interpreter Lock (GIL) on some cython operations, see [here](#)
- Plotting methods are now available as attributes of the `.plot` accessor, see [here](#)
- The sorting API has been revamped to remove some long-time inconsistencies, see [here](#)
- Support for a `datetime64[ns]` with timezones as a first-class dtype, see [here](#)
- The default for `to_datetime` will now be to `raise` when presented with unparseable formats, previously this would return the original input. Also, date parse functions now return consistent results. See [here](#)
- The default for `dropna` in `HDFStore` has changed to `False`, to store by default all rows even if they are all `NaN`, see [here](#)
- Datetime accessor (`dt`) now supports `Series.dt.strftime` to generate formatted strings for datetime-likes, and `Series.dt.total_seconds` to generate each duration of the `timedelta` in seconds. See [here](#)
- `Period` and `PeriodIndex` can handle multiplied freq like `3D`, which corresponding to 3 days span. See [here](#)
- Development installed versions of pandas will now have PEP440 compliant version strings ([GH9518](#))
- Development support for benchmarking with the [Air Speed Velocity](#) library ([GH8316](#))

- Support for reading SAS xport files, see [here](#)
- Documentation comparing SAS to *pandas*, see [here](#)
- Removal of the automatic TimeSeries broadcasting, deprecated since 0.8.0, see [here](#)
- Display format with plain text can optionally align with Unicode East Asian Width, see [here](#)
- Compatibility with Python 3.5 ([GH11097](#))
- Compatibility with matplotlib 1.5.0 ([GH11111](#))

See the [v0.17.0 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.17.0.

### 38.14.1 Thanks

- Alex Rothberg
- Andrea Bedini
- Andrew Rosenfeld
- Andy Li
- Anthonios Partheniou
- Artemy Kolchinsky
- Bernard Willers
- Charlie Clark
- Chris
- Chris Whelan
- Christoph Gohlke
- Christopher Whelan
- Clark Fitzgerald
- Clearfield Christopher
- Dan Ringwalt
- Daniel Ni
- Data & Code Expert Experimenting with Code on Data
- David Cottrell
- David John Gagne
- David Kelly
- ETF
- Eduardo Schettino
- Egor
- Egor Panfilov
- Evan Wright
- Frank Pinter
- Gabriel Araujo

- Garrett-R
- Gianluca Rossi
- Guillaume Gay
- Guillaume Poulin
- Harsh Nisar
- Ian Henriksen
- Ian Hoegen
- Jaidev Deshpande
- Jan Rudolph
- Jan Schulz
- Jason Swails
- Jeff Reback
- Jonas Buyl
- Joris Van den Bossche
- Joris Vankerschaver
- Josh Levy-Kramer
- Julien Danjou
- Ka Wo Chen
- Karrie Kehoe
- Kelsey Jordahl
- Kerby Shedden
- Kevin Sheppard
- Lars Buitinck
- Leif Johnson
- Luis Ortiz
- Mac
- Matt Gambogi
- Matt Savoie
- Matthew Gilbert
- Maximilian Roos
- Michelangelo D'Agostino
- Mortada Mehyar
- Nick Eubank
- Nipun Batra
- Ondřej Čertík
- Phillip Cloud

- Pratap Vardhan
- Rafal Skolasinski
- Richard Lewis
- Rinoc Johnson
- Rob Levy
- Robert Gieseke
- Safia Abdalla
- Samuel Denny
- Saumitra Shahapure
- Sebastian Pölsterl
- Sebastian Rubbert
- Sheppard, Kevin
- Sinhrks
- Siu Kwan Lam
- Skipper Seabold
- Spencer Carrucciu
- Stephan Hoyer
- Stephen Hoover
- Stephen Pascoe
- Terry Santegoeds
- Thomas Grainger
- Tjerk Santegoeds
- Tom Augspurger
- Vincent Davis
- Winterflower
- Yaroslav Halchenko
- Yuan Tang (Terry)
- agijsberts
- ajcr
- behzad nouri
- cel4
- cyrusmaher
- davidovitch
- ganego
- jreback
- juricast



- larvian
- maximilianr
- msund
- rekcahpassyla
- robertzk
- scls19fr
- seth-p
- sinhrks
- springcoil
- terrytangyuan
- tzinckgraf

## 38.15 pandas 0.16.2

**Release date:** (June 12, 2015)

This is a minor release from 0.16.1 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements.

Highlights include:

- A new `pipe` method, see [here](#)
- Documentation on how to use `numba` with `pandas`, see [here](#)

See the [v0.16.2 Whatsnew](#) overview for an extensive list of all enhancements and bugs that have been fixed in 0.16.2.

### 38.15.1 Thanks

- Andrew Rosenfeld
- Artemy Kolchinsky
- Bernard Willers
- Christer van der Meeren
- Christian Hudon
- Constantine Glen Evans
- Daniel Julius Lasiman
- Evan Wright
- Francesco Brundu
- Gaëtan de Menten
- Jake VanderPlas
- James Hiebert
- Jeff Reback

- Joris Van den Bossche
- Justin Lecher
- Ka Wo Chen
- Kevin Sheppard
- Mortada Mehyar
- Morton Fox
- Robin Wilson
- Thomas Grainger
- Tom Ajanian
- Tom Augspurger
- Yoshiki Vázquez Baeza
- Younggun Kim
- austinc
- behzad nouri
- jreback
- lexical
- rekcahpassyla
- scls19fr
- sinhrks

## **38.16 pandas 0.16.1**

**Release date:** (May 11, 2015)

This is a minor release from 0.16.0 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. A small number of API changes were necessary to fix existing bugs.

See the [v0.16.1 Whatsnew](#) overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.16.1.

### **38.16.1 Thanks**

- Alfonso MHC
- Andy Hayden
- Artemy Kolchinsky
- Chris Gilmer
- Chris Grinolds
- Dan Birken
- David BROCHART
- David Hirschfeld

- David Stephens
- Dr. Leo
- Evan Wright
- Frans van Dunné
- Hatem Nassrat
- Henning Sperr
- Hugo Herter
- Jan Schulz
- Jeff Blackburne
- Jeff Reback
- Jim Crist
- Jonas Abernot
- Joris Van den Bossche
- Kerby Shedden
- Leo Razoumov
- Manuel Riel
- Mortada Mehyar
- Nick Burns
- Nick Eubank
- Olivier Grisel
- Phillip Cloud
- Pietro Battiston
- Roy Hyunjin Han
- Sam Zhang
- Scott Sanderson
- Stephan Hoyer
- Tiago Antao
- Tom Ajamian
- Tom Augspurger
- Tomaz Berisa
- Vikram Shirgur
- Vladimir Filimonov
- William Hogman
- Yasin A
- Younggun Kim
- behzad nouri

- dsm054
- floydsoft
- flying-sheep
- gfr
- jnmclarty
- jreback
- ksanghai
- lucas
- mschmohl
- ptype
- rockg
- scls19fr
- sinhrks

## 38.17 pandas 0.16.0

**Release date:** (March 22, 2015)

This is a major release from 0.15.2 and includes a number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

- `DataFrame.assign` method, see [here](#)
- `Series.to_coo/from_coo` methods to interact with `scipy.sparse`, see [here](#)
- Backwards incompatible change to `Timedelta` to conform the `.seconds` attribute with `datetime.timedelta`, see [here](#)
- Changes to the `.loc` slicing API to conform with the behavior of `.ix` see [here](#)
- Changes to the default for ordering in the `Categorical` constructor, see [here](#)
- The `pandas.tools.rplot`, `pandas.sandbox.qtpandas` and `pandas.rpy` modules are deprecated. We refer users to external packages like [seaborn](#), [pandas-qt](#) and [rpy2](#) for similar or equivalent functionality, see [here](#)

See the [v0.16.0 Whatsnew](#) overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.16.0.

### 38.17.1 Thanks

- Aaron Toth
- Alan Du
- Alessandro Amici
- Artemy Kolchinsky
- Ashwini Chaudhary

- Ben Schiller
- Bill Letson
- Brandon Bradley
- Chau Hoang
- Chris Reynolds
- Chris Whelan
- Christer van der Meeren
- David Cottrell
- David Stephens
- Ehsan Azarnasab
- Garrett-R
- Guillaume Gay
- Jake Torcasso
- Jason Sexauer
- Jeff Reback
- John McNamara
- Joris Van den Bossche
- Joschka zur Jacobsmühlen
- Juarez Bochi
- Junya Hayashi
- K.-Michael Aye
- Kerby Shedden
- Kevin Sheppard
- Kieran O'Mahony
- Kodi Arfer
- Matti Airas
- Min RK
- Mortada Mehyar
- Robert
- Scott E Lasley
- Scott Lasley
- Sergio Pascual
- Skipper Seabold
- Stephan Hoyer
- Thomas Grainger
- Tom Augspurger

- TomAugspurger
- Vladimir Filimonov
- Vyomkesh Tripathi
- Will Holmgren
- Yulong Yang
- behzad nouri
- bertrandhaut
- bjonen
- cel4
- clham
- hsperr
- ischwabacher
- jnmclarty
- josham
- jreback
- omtinez
- roch
- sinhrks
- unutbu

## 38.18 pandas 0.15.2

**Release date:** (December 12, 2014)

This is a minor release from 0.15.1 and includes a large number of bug fixes along with several new features, enhancements, and performance improvements. A small number of API changes were necessary to fix existing bugs.

See the [v0.15.2 Whatsnew](#) overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.2.

### 38.18.1 Thanks

- Aaron Staple
- Angelos Evripiotis
- Artemy Kolchinsky
- Benoit Pointet
- Brian Jacobowski
- Charalampos Papaloizou
- Chris Warth
- David Stephens

- Fabio Zanini
- Francesc Via
- Henry Kleynhans
- Jake VanderPlas
- Jan Schulz
- Jeff Reback
- Jeff Tratner
- Joris Van den Bossche
- Kevin Sheppard
- Matt Suggit
- Matthew Brett
- Phillip Cloud
- Rupert Thompson
- Scott E Lasley
- Stephan Hoyer
- Stephen Simmons
- Sylvain Corlay
- Thomas Grainger
- Tiago Antao
- Trent Hauck
- Victor Chaves
- Victor Salgado
- Vikram Bhandoh
- WANG Aiyong
- Will Holmgren
- behzad nouri
- broessli
- charalampos papaloizou
- immerrr
- jnmclarty
- jreback
- mgilbert
- onesandzeroes
- peadarcoyle
- rockg
- seth-p

- sinhrks
- unutbu
- wavedatalab
- Åsmund Hjulstad

## 38.19 pandas 0.15.1

**Release date:** (November 9, 2014)

This is a minor release from 0.15.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

See the [v0.15.1 Whatsnew](#) overview for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.1.

### 38.19.1 Thanks

- Aaron Staple
- Andrew Rosenfeld
- Anton I. Sipos
- Artemy Kolchinsky
- Bill Letson
- Dave Hughes
- David Stephens
- Guillaume Horel
- Jeff Reback
- Joris Van den Bossche
- Kevin Sheppard
- Nick Stahl
- Sanghee Kim
- Stephan Hoyer
- TomAugspurger
- WANG Aiyong
- behzad nouri
- immerrr
- jnmclarty
- jreback
- pallav-fdsi
- unutbu



## 38.20 pandas 0.15.0

**Release date:** (October 18, 2014)

This is a major release from 0.14.1 and includes a number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

- Drop support for NumPy < 1.7.0 ([GH7711](#))
- The `Categorical` type was integrated as a first-class pandas type, see [here](#)
- New scalar type `Timedelta`, and a new index type `TimedeltaIndex`, see [here](#)
- New DataFrame default display for `df.info()` to include memory usage, see [Memory Usage](#)
- New datetimelike properties accessor `.dt` for Series, see [Datetimelike Properties](#)
- Split indexing documentation into [Indexing and Selecting Data](#) and [MultiIndex / Advanced Indexing](#)
- Split out string methods documentation into [Working with Text Data](#)
- `read_csv` will now by default ignore blank lines when parsing, see [here](#)
- API change in using Indexes in set operations, see [here](#)
- Internal refactoring of the `Index` class to no longer sub-class `ndarray`, see [Internal Refactoring](#)
- dropping support for PyTables less than version 3.0.0, and numexpr less than version 2.1 ([GH7990](#))

See the [v0.15.0 Whatsnew](#) overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.15.0.

### 38.20.1 Thanks

- Aaron Schumacher
- Adam Greenhall
- Andy Hayden
- Anthony O'Brien
- Artemy Kolchinsky
- behzad nouri
- Benedikt Sauer
- benjamin
- Benjamin Thyreau
- Ben Schiller
- bjonon
- BorisVerk
- Chris Reynolds
- Chris Stoafer
- Dav Clark
- dlovel

- DSM
- dsm054
- FragLegs
- German Gomez-Herrero
- Hsiaoming Yang
- Huan Li
- hunterowens
- Hyungtae Kim
- immerrr
- Isaac Slavitt
- ischwabacher
- Jacob Schaer
- Jacob Wasserman
- Jan Schulz
- Jeff Tratner
- Jesse Farnham
- jmorris0x0
- jnmclarty
- Joe Bradish
- Joerg Rittinger
- John W. O'Brien
- Joris Van den Bossche
- jreback
- Kevin Sheppard
- klonuo
- Kyle Meyer
- lexical
- Max Chang
- mcjcode
- Michael Mueller
- Michael W Schatzow
- Mike Kelly
- Mortada Mehyar
- mtrbean
- Nathan Sanders
- Nathan Typanski

- onesandzeroes
- Paul Masurel
- Phillip Cloud
- Pietro Battiston
- RenzoBertocchi
- rockg
- Ross Petchler
- seth-p
- Shahul Hameed
- Shashank Agarwal
- sinhrks
- someben
- stahlous
- stas-sl
- Stephan Hoyer
- thatneat
- tom-alcorn
- TomAugspurger
- Tom Augspurger
- Tony Lorenzo
- unknown
- unutbu
- Wes Turner
- Wilfred Hughes
- Yevgeniy Grechka
- Yoshiki Vázquez Baeza
- zachcp

## 38.21 pandas 0.14.1

**Release date:** (July 11, 2014)

This is a minor release from 0.14.0 and includes a small number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

- New methods `select_dtypes()` to select columns based on the dtype and `sem()` to calculate the standard error of the mean.
- Support for dateutil timezones (see [docs](#)).

- Support for ignoring full line comments in the `read_csv()` text parser.
- New documentation section on *Options and Settings*.
- Lots of bug fixes.

See the [v0.14.1 Whatsnew](#) overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.14.1.

### 38.21.1 Thanks

- Andrew Rosenfeld
- Andy Hayden
- Benjamin Adams
- Benjamin M. Gross
- Brian Quistorff
- Brian Wignall
- bwignall
- clham
- Daniel Waeber
- David Bew
- David Stephens
- DSM
- dsm054
- helger
- immerrr
- Jacob Schaer
- jaimefrio
- Jan Schulz
- John David Reaver
- John W. O'Brien
- Joris Van den Bossche
- jreback
- Julien Danjou
- Kevin Sheppard
- K.-Michael Aye
- Kyle Meyer
- lexical
- Matthew Brett
- Matt Wittmann

- Michael Mueller
- Mortada Mehyar
- onesandzeroes
- Phillip Cloud
- Rob Levy
- rockg
- sanguineturtle
- Schaer, Jacob C
- seth-p
- sinhrks
- Stephan Hoyer
- Thomas Kluyver
- Todd Jennings
- TomAugspurger
- unknown
- yelite

## 38.22 pandas 0.14.0

**Release date:** (May 31, 2014)

This is a major release from 0.13.1 and includes a number of API changes, several new features, enhancements, and performance improvements along with a large number of bug fixes.

Highlights include:

- Officially support Python 3.4
- SQL interfaces updated to use `sqlalchemy`, see [here](#).
- Display interface changes, see [here](#)
- MultiIndexing using Slicers, see [here](#).
- Ability to join a singly-indexed DataFrame with a multi-indexed DataFrame, see [here](#)
- More consistency in groupby results and more flexible groupby specifications, see [here](#)
- Holiday calendars are now supported in `CustomBusinessDay`, see [here](#)
- Several improvements in plotting functions, including: hexbin, area and pie plots, see [here](#).
- Performance doc section on I/O operations, see [here](#)

See the [v0.14.0 Whatsnew](#) overview or the issue tracker on GitHub for an extensive list of all API changes, enhancements and bugs that have been fixed in 0.14.0.

### **38.22.1 Thanks**

- Acanthostega
- Adam Marcus
- agijsberts
- akittredge
- Alex Gaudio
- Alex Rothberg
- AllenDowney
- Andrew Rosenfeld
- Andy Hayden
- ankostis
- anomrake
- Antoine Mazières
- anton-d
- bashtage
- Benedikt Sauer
- benjamin
- Brad Buran
- bwignall
- cgohlke
- chebee7i
- Christopher Whelan
- Clark Fitzgerald
- clham
- Dale Jung
- Dan Allan
- Dan Birken
- danielballan
- Daniel Waeber
- David Jung
- David Stephens
- Douglas McNeil
- DSM
- Garrett Drapala
- Gouthaman Balaraman
- Guillaume Poulin

- hshimizu77
- hugo
- immerrr
- ischwabacher
- Jacob Howard
- Jacob Schaer
- jaimefrio
- Jason Sexauer
- Jeff Reback
- Jeffrey Starr
- Jeff Tratner
- John David Reaver
- John McNamara
- John W. O'Brien
- Jonathan Chambers
- Joris Van den Bossche
- jreback
- jsexauer
- Julia Evans
- Júlio
- Katie Atkinson
- kdiether
- Kelsey Jordahl
- Kevin Sheppard
- K.-Michael Aye
- Matthias Kuhn
- Matt Wittmann
- Max Grender-Jones
- Michael E. Gruen
- michaelws
- mikebailey
- Mike Kelly
- Nipun Batra
- Noah Spies
- ojdo
- onesandzeroes

- Patrick O’Keeffe
- phaebz
- Phillip Cloud
- Pietro Battiston
- PKEuS
- Randy Carnevale
- ribonooous
- Robert Gibboni
- rockg
- sinhrks
- Skipper Seabold
- SplashDance
- Stephan Hoyer
- Tim Cera
- Tobias Brandt
- Todd Jennings
- TomAugspurger
- Tom Augspurger
- unutbu
- westurner
- Yaroslav Halchenko
- y-p
- zach powers

## 38.23 pandas 0.13.1

**Release date:** (February 3, 2014)

### 38.23.1 New Features

- Added `date_format` and `datetime_format` attribute to `ExcelWriter`. ([GH4133](#))

### 38.23.2 API Changes

- `Series.sort` will raise a `ValueError` (rather than a `TypeError`) on sorting an object that is a view of another ([GH5856](#), [GH5853](#))
- Raise/Warn `SettingWithCopyError` (according to the option `chained_assignment` in more cases, when detecting chained assignment, related ([GH5938](#), [GH6025](#)))



- `DataFrame.head(0)` returns self instead of empty frame (GH5846)
- `autocorrelation_plot` now accepts `**kwargs`. (GH5623)
- `convert_objects` now accepts a `convert_timedeltas='coerce'` argument to allow forced dtype conversion of timedeltas (GH5458, :issue:5689)
- Add `-NaN` and `-nan` to the default set of NA values (GH5952). See *NA Values*.
- `NDFrame` now has an `equals` method. (GH5283)
- `DataFrame.apply` will use the `reduce` argument to determine whether a `Series` or a `DataFrame` should be returned when the `DataFrame` is empty (GH6007).

### 38.23.3 Experimental Features

#### 38.23.4 Improvements to existing features

- perf improvements in `Series` datetime/timedelta binary operations (GH5801)
- `option_context` context manager now available as top-level API (GH5752)
- `df.info()` view now display dtype info per column (GH5682)
- `df.info()` now honors option `max_info_rows`, disable null counts for large frames (GH5974)
- perf improvements in `DataFrame.count/dropna` for `axis=1`
- `Series.str.contains` now has a `regex=False` keyword which can be faster for plain (non-regex) string patterns. (GH5879)
- support dtypes property on `Series/Panel/Panel4D`
- extend `Panel.apply` to allow arbitrary functions (rather than only ufuncs) (GH1148) allow multiple axes to be used to operate on slabs of a `Panel`
- The `ArrayFormatter` for datetime and `timedelta64` now intelligently limit precision based on the values in the array (GH3401)
- `pd.show_versions()` is now available for convenience when reporting issues.
- perf improvements to `Series.str.extract` (GH5944)
- perf improvements in `dtypes/ftypes` methods (GH5968)
- perf improvements in indexing with object dtypes (GH5968)
- improved dtype inference for `timedelta` like passed to constructors (GH5458, GH5689)
- escape special characters when writing to latex (:issue: 5374)
- perf improvements in `DataFrame.apply` (GH6013)
- `pd.read_csv` and `pd.to_datetime` learned a new `infer_datetime_format` keyword which greatly improves parsing perf in many cases. Thanks to @lexical for suggesting and @danbirken for rapidly implementing. (GH5490, :issue:6021)
- add ability to recognize '%p' format code (am/pm) to date parsers when the specific format is supplied (GH5361)
- Fix performance regression in JSON IO (GH5765)
- performance regression in Index construction from `Series` (GH6150)

### 38.23.5 Bug Fixes

- Bug in `io.wb.get_countries` not including all countries (GH6008)
- Bug in Series replace with timestamp dict (GH5797)
- `read_csv/read_table` now respects the *prefix* kwarg (GH5732).
- Bug in selection with missing values via `.ix` from a duplicate indexed DataFrame failing (GH5835)
- Fix issue of boolean comparison on empty DataFrames (GH5808)
- Bug in `isnull` handling NaT in an object array (GH5443)
- Bug in `to_datetime` when passed a `np.nan` or integer datelike and a format string (GH5863)
- Bug in groupby dtype conversion with datetimelike (GH5869)
- Regression in handling of empty Series as indexers to Series (GH5877)
- Bug in internal caching, related to (GH5727)
- Testing bug in reading JSON/msgpack from a non-filepath on windows under py3 (GH5874)
- Bug when assigning to `.ix[tuple(...)]` (GH5896)
- Bug in fully reindexing a Panel (GH5905)
- Bug in `idxmin/max` with object dtypes (GH5914)
- Bug in `BusinessDay` when adding `n` days to a date not on offset when `n>5` and `n%5==0` (GH5890)
- Bug in assigning to chained series with a series via `ix` (GH5928)
- Bug in creating an empty DataFrame, copying, then assigning (GH5932)
- Bug in `DataFrame.tail` with empty frame (GH5846)
- Bug in propagating metadata on `resample` (GH5862)
- Fixed string-representation of NaT to be “NaT” (GH5708)
- Fixed string-representation for Timestamp to show nanoseconds if present (GH5912)
- `pd.match` not returning passed sentinel
- `Panel.to_frame()` no longer fails when `major_axis` is a `MultiIndex` (GH5402).
- Bug in `pd.read_msgpack` with inferring a `DateTimeIndex` frequency incorrectly (GH5947)
- Fixed `to_datetime` for array with both Tz-aware datetimes and NaT’s (GH5961)
- Bug in rolling skew/kurtosis when passed a Series with bad data (GH5749)
- Bug in `scipy.interpolate` methods with a datetime index (GH5975)
- Bug in NaT comparison if a mixed `datetime/np.datetime64` with NaT were passed (GH5968)
- Fixed bug with `pd.concat` losing dtype information if all inputs are empty (GH5742)
- Recent changes in IPython cause warnings to be emitted when using previous versions of pandas in QtConsole, now fixed. If you’re using an older version and need to suppress the warnings, see (GH5922).
- Bug in merging `timedelta` dtypes (GH5695)
- Bug in `plotting.scatter_matrix` function. Wrong alignment among diagonal and off-diagonal plots, see (GH5497).
- Regression in Series with a multi-index via `ix` (GH6018)

- Bug in `Series.xs` with a multi-index (GH6018)
- Bug in Series construction of mixed type with datelike and an integer (which should result in object type and not automatic conversion) (GH6028)
- Possible segfault when chained indexing with an object array under NumPy 1.7.1 (GH6026, GH6056)
- Bug in setting using fancy indexing a single element with a non-scalar (e.g. a list), (GH6043)
- `to_sql` did not respect `if_exists` (GH4110 GH4304)
- Regression in `.get(None)` indexing from 0.12 (GH5652)
- Subtle `iloc` indexing bug, surfaced in (GH6059)
- Bug with insert of strings into `DatetimeIndex` (GH5818)
- Fixed unicode bug in `to_html/HTML` repr (GH6098)
- Fixed missing arg validation in `get_options_data` (GH6105)
- Bug in assignment with duplicate columns in a frame where the locations are a slice (e.g. next to each other) (GH6120)
- Bug in propagating `_ref_locs` during construction of a `DataFrame` with dups index/columns (GH6121)
- Bug in `DataFrame.apply` when using mixed datelike reductions (GH6125)
- Bug in `DataFrame.append` when appending a row with different columns (GH6129)
- Bug in `DataFrame` construction with recarray and non-ns datetime dtype (GH6140)
- Bug in `.loc` setitem indexing with a dataframe on rhs, multiple item setting, and a datetimelike (GH6152)
- Fixed a bug in `query/eval` during lexicographic string comparisons (GH6155).
- Fixed a bug in `query` where the index of a single-element `Series` was being thrown away (GH6148).
- Bug in `HDFStore` on appending a dataframe with multi-indexed columns to an existing table (GH6167)
- Consistency with dtypes in setting an empty `DataFrame` (GH6171)
- Bug in selecting on a multi-index `HDFStore` even in the presence of under specified column spec (GH6169)
- Bug in `nanops.var` with `ddof=1` and 1 elements would sometimes return `inf` rather than `nan` on some platforms (GH6136)
- Bug in Series and DataFrame bar plots ignoring the `use_index` keyword (GH6209)
- Bug in groupby with mixed str/int under python3 fixed; `argsort` was failing (GH6212)

## 38.24 pandas 0.13.0

**Release date:** January 3, 2014

### 38.24.1 New Features

- `plot(kind='kde')` now accepts the optional parameters `bw_method` and `ind`, passed to `scipy.stats.gaussian_kde()` (for `scipy >= 0.11.0`) to set the bandwidth, and to `gkde.evaluate()` to specify the indices at which it is evaluated, respectively. See `scipy` docs. (GH4298)
- Added `isin` method to `DataFrame` (GH4211)

- `df.to_clipboard()` learned a new `excel` keyword that let's you paste df data directly into excel (enabled by default). (GH5070).
- Clipboard functionality now works with PySide (GH4282)
- New `extract` string method returns regex matches more conveniently (GH4685)
- Auto-detect field widths in `read_fwf` when unspecified (GH4488)
- `to_csv()` now outputs datetime objects according to a specified format string via the `date_format` keyword (GH4313)
- Added `LastWeekOfMonth` `DateOffset` (GH4637)
- Added `cumcount` `groupby` method (GH4646)
- Added `FY5253`, and `FY5253Quarter` `DateOffsets` (GH4511)
- Added `mode()` method to `Series` and `DataFrame` to get the statistical mode(s) of a column/series. (GH5367)

### 38.24.2 Experimental Features

- The new `eval()` function implements expression evaluation using `numexpr` behind the scenes. This results in large speedups for complicated expressions involving large `DataFrames`/`Series`.
- `DataFrame` has a new `eval()` that evaluates an expression in the context of the `DataFrame`; allows inline expression assignment
- A `query()` method has been added that allows you to select elements of a `DataFrame` using a natural query syntax nearly identical to Python syntax.
- `pd.eval` and friends now evaluate operations involving `datetime64` objects in Python space because `numexpr` cannot handle `NaT` values (GH4897).
- Add `msgpack` support via `pd.read_msgpack()` and `pd.to_msgpack()` / `df.to_msgpack()` for serialization of arbitrary pandas (and python objects) in a lightweight portable binary format (GH686, GH5506)
- Added PySide support for the `qtandas DataFrameModel` and `DataFrameWidget`.
- Added `pandas.io.gbq` for reading from (and writing to) Google BigQuery into a `DataFrame`. (GH4140)

### 38.24.3 Improvements to existing features

- `read_html` now raises a `URLError` instead of catching and raising a `ValueError` (GH4303, GH4305)
- `read_excel` now supports an integer in its `sheetname` argument giving the index of the sheet to read in (GH4301).
- `get_dummies` works with `NaN` (GH4446)
- Added a test for `read_clipboard()` and `to_clipboard()` (GH4282)
- Added `bins` argument to `value_counts` (GH3945), also `sort` and `ascending`, now available in `Series` method as well as top-level function.
- Text parser now treats anything that reads like `inf` ("`inf`", "`Inf`", "`-Inf`", "`iNf`", etc.) to infinity. (GH4220, GH4219), affecting `read_table`, `read_csv`, etc.
- Added a more informative error message when plot arguments contain overlapping color and style arguments (GH4402)
- Significant table writing performance improvements in `HDFStore`

- JSON date serialization now performed in low-level C code.
- JSON support for encoding `datetime.time`
- Expanded JSON docs, more info about orient options and the use of the `numpy` param when decoding.
- Add `drop_level` argument to `xs` ([GH4180](#))
- Can now resample a `DataFrame` with `ohlc` ([GH2320](#))
- `Index.copy()` and `MultiIndex.copy()` now accept keyword arguments to change attributes (i.e., `names`, `levels`, `labels`) ([GH4039](#))
- Add `rename` and `set_names` methods to `Index` as well as `set_names`, `set_levels`, `set_labels` to `MultiIndex`. ([GH4039](#)) with improved validation for all ([GH4039](#), [GH4794](#))
- A `Series` of dtype `timedelta64[ns]` can now be divided/multiplied by an integer series ([GH4521](#))
- A `Series` of dtype `timedelta64[ns]` can now be divided by another `timedelta64[ns]` object to yield a `float64` typed `Series`. This is frequency conversion; astyping is also supported.
- `Timedelta64` support `fillna/ffill/bfill` with an integer interpreted as seconds, or a `timedelta` ([GH3371](#))
- Box numeric ops on `timedelta` `Series` ([GH4984](#))
- `Datetime64` support `ffill/bfill`
- Performance improvements with `__getitem__` on `DataFrames` with when the key is a column
- Support for using a `DatetimeIndex/PeriodsIndex` directly in a datelike calculation e.g. `s.s.index` ([GH4629](#))
- Better/cleaned up exceptions in `core/common`, `io/excel` and `core/format` ([GH4721](#), [GH3954](#)), as well as cleaned up test cases in `tests/test_frame`, `tests/test_multilevel` ([GH4732](#)).
- Performance improvement of timeseries plotting with `PeriodIndex` and added test to `vbench` ([GH4705](#) and [GH4722](#))
- Add `axis` and `level` keywords to `where`, so that the `other` argument can now be an alignable pandas object.
- `to_datetime` with a format of `'%Y%m%d'` now parses much faster
- It's now easier to hook new Excel writers into pandas (just subclass `ExcelWriter` and register your engine). You can specify an engine in `to_excel` or in `ExcelWriter`. You can also specify which writers you want to use by default with config options `io.excel.xlsx.writer` and `io.excel.xls.writer`. ([GH4745](#), [GH4750](#))
- `Panel.to_excel()` now accepts keyword arguments that will be passed to its `DataFrame's to_excel()` methods. ([GH4750](#))
- Added `XlsxWriter` as an optional `ExcelWriter` engine. This is about 5x faster than the default `openpyxl` `xlsx` writer and is equivalent in speed to the `xlwt` `xls` writer module. ([GH4542](#))
- allow `DataFrame` constructor to accept more list-like objects, e.g. `list of collections.Sequence` and `array.Array` objects ([GH3783](#), [GH4297](#), [GH4851](#)), thanks @lgautier
- `DataFrame` constructor now accepts a NumPy masked record array ([GH3478](#)), thanks @jnothman
- `__getitem__` with tuple key (e.g., `[:, 2]`) on `Series` without `MultiIndex` raises `ValueError` ([GH4759](#), [GH4837](#))
- `read_json` now raises a (more informative) `ValueError` when the dict contains a bad key and `orient='split'` ([GH4730](#), [GH4838](#))

- `read_stata` now accepts Stata 13 format ([GH4291](#))
- `ExcelWriter` and `ExcelFile` can be used as contextmanagers. ([GH3441](#), [GH4933](#))
- `pandas` is now tested with two different versions of `statsmodels` (0.4.3 and 0.5.0) ([GH4981](#)).
- Better string representations of `MultiIndex` (including ability to roundtrip via `repr`). ([GH3347](#), [GH4935](#))
- Both `ExcelFile` and `read_excel` to accept an `xlrd.Book` for the `io` (formerly `path_or_buf`) argument; this requires engine to be set. ([GH4961](#)).
- `concat` now gives a more informative error message when passed objects that cannot be concatenated ([GH4608](#)).
- Add `halflife` option to exponentially weighted moving functions (PR [GH4998](#))
- `to_dict` now takes `records` as a possible outtype. Returns an array of column-keyed dictionaries. ([GH4936](#))
- `tz_localize` can infer a fall daylight savings transition based on the structure of unlocalized data ([GH4230](#))
- `DatetimeIndex` is now in the API documentation
- Improve support for converting R datasets to pandas objects (more informative index for timeseries and numeric, support for factors, dist, and high-dimensional arrays).
- `read_html()` now supports the `parse_dates`, `tupleize_cols` and `thousands` parameters ([GH4770](#)).
- `json_normalize()` is a new method to allow you to create a flat table from semi-structured JSON data. *See the docs* ([GH1067](#))
- `DataFrame.from_records()` will now accept generators ([GH4910](#))
- `DataFrame.interpolate()` and `Series.interpolate()` have been expanded to include interpolation methods from `scipy`. ([GH4434](#), [GH1892](#))
- `Series` now supports a `to_frame` method to convert it to a single-column `DataFrame` ([GH5164](#))
- `DatetimeIndex` (and `date_range`) can now be constructed in a left- or right-open fashion using the `closed` parameter ([GH4579](#))
- Python csv parser now supports `usecols` ([GH4335](#))
- Added support for Google Analytics v3 API segment IDs that also supports v2 IDs. ([GH5271](#))
- `NDFrame.drop()` now accepts names as well as integers for the `axis` argument. ([GH5354](#))
- Added short docstrings to a few methods that were missing them + fixed the docstrings for Panel flex methods. ([GH5336](#))
- `NDFrame.drop()`, `NDFrame.dropna()`, and `.drop_duplicates()` all accept `inplace` as a keyword argument; however, this only means that the wrapper is updated inplace, a copy is still made internally. ([GH1960](#), [GH5247](#), [GH5628](#), and related [GH2325](#) [still not closed])
- Fixed bug in `tools.plotting.andrews_curves` so that lines are drawn grouped by color as expected.
- `read_excel()` now tries to convert integral floats (like `1.0`) to `int` by default. ([GH5394](#))
- Excel writers now have a default option `merge_cells` in `to_excel()` to merge cells in `MultiIndex` and Hierarchical Rows. Note: using this option it is no longer possible to round trip Excel files with merged `MultiIndex` and Hierarchical Rows. Set the `merge_cells` to `False` to restore the previous behaviour. ([GH5254](#))
- The FRED `DataReader` now accepts multiple series (:issue‘3413’)
- `StataWriter` adjusts variable names to Stata’s limitations ([GH5709](#))

### 38.24.4 API Changes

- `DataFrame.reindex()` and forward/backward filling now raises `ValueError` if either index is not monotonic ([GH4483](#), [GH4484](#)).
- pandas now is Python 2/3 compatible without the need for 2to3 thanks to @jtratrner. As a result, pandas now uses iterators more extensively. This also led to the introduction of substantive parts of the Benjamin Peterson's `six` library into `compat`. ([GH4384](#), [GH4375](#), [GH4372](#))
- `pandas.util.compat` and `pandas.util.py3compat` have been merged into `pandas.compat`. `pandas.compat` now includes many functions allowing 2/3 compatibility. It contains both list and iterator versions of `range`, `filter`, `map` and `zip`, plus other necessary elements for Python 3 compatibility. `lmap`, `lzip`, `lrange` and `lfilter` all produce lists instead of iterators, for compatibility with `numpy`, subscripting and pandas constructors. ([GH4384](#), [GH4375](#), [GH4372](#))
- deprecated `iterkv`, which will be removed in a future release (was just an alias of `iteritems` used to get around 2to3's changes). ([GH4384](#), [GH4375](#), [GH4372](#))
- `Series.get` with negative indexers now returns the same as `[]` ([GH4390](#))
- allow `ix/loc` for `Series/DataFrame/Panel` to set on any axis even when the single-key is not currently contained in the index for that axis ([GH2578](#), [GH5226](#), [GH5632](#), [GH5720](#), [GH5744](#), [GH5756](#))
- Default export for `to_clipboard` is now `csv` with a sep of `t` for `compat` ([GH3368](#))
- `at` now will enlarge the object inplace (and return the same) ([GH2578](#))
- `DataFrame.plot` will scatter plot `x` versus `y` by passing `kind='scatter'` ([GH2215](#))
- `HDFStore`
  - `append_to_multiple` automatically synchronizes writing rows to multiple tables and adds a `dropna` kwarg ([GH4698](#))
  - handle a passed `Series` in table format ([GH4330](#))
  - added an `is_open` property to indicate if the underlying file handle is `is_open`; a closed store will now report 'CLOSED' when viewing the store (rather than raising an error) ([GH4409](#))
  - a close of a `HDFStore` now will close that instance of the `HDFStore` but will only close the actual file if the ref count (by `PyTables`) w.r.t. all of the open handles are 0. Essentially you have a local instance of `HDFStore` referenced by a variable. Once you close it, it will report closed. Other references (to the same file) will continue to operate until they themselves are closed. Performing an action on a closed file will raise `ClosedFileError`
  - removed the `_quiet` attribute, replace by a `DuplicateWarning` if retrieving duplicate rows from a table ([GH4367](#))
  - removed the `warn` argument from `open`. Instead a `PossibleDataLossError` exception will be raised if you try to use `mode='w'` with an `OPEN` file handle ([GH4367](#))
  - allow a passed locations array or mask as a `where` condition ([GH4467](#))
  - add the keyword `dropna=True` to `append` to change whether ALL nan rows are not written to the store (default is `True`, ALL nan rows are NOT written), also settable via the option `io.hdf.dropna_table` ([GH4625](#))
  - the `format` keyword now replaces the `table` keyword; allowed values are `fixed(f) | table(t)` the `Storer` format has been renamed to `Fixed`
  - a column multi-index will be recreated properly ([GH4710](#)); raise on trying to use a multi-index with `data_columns` on the same axis
  - `select_as_coordinates` will now return an `Int64Index` of the resultant selection set



- support `timedelta64[ns]` as a serialization type (GH3577)
- store `datetime.date` objects as ordinals rather than `timetuples` to avoid timezone issues (GH2852), thanks @tavistmorph and @numband
- `numexpr` 2.2.2 fixes incompatibility in `PyTables` 2.4 (GH4908)
- `flush` now accepts an `fsync` parameter, which defaults to `False` (GH5364)
- `unicode` indices not supported on `table` formats (GH5386)
- pass thru store creation arguments; can be used to support in-memory stores
- JSON
  - added `date_unit` parameter to specify resolution of timestamps. Options are seconds, milliseconds, microseconds and nanoseconds. (GH4362, GH4498).
  - added `default_handler` parameter to allow a callable to be passed which will be responsible for handling otherwise unserializable objects. (GH5138)
- Index and MultiIndex changes (GH4039):
  - Setting `levels` and `labels` directly on `MultiIndex` is now deprecated. Instead, you can use the `set_levels()` and `set_labels()` methods.
  - `levels`, `labels` and `names` properties no longer return lists, but instead return containers that do not allow setting of items ('mostly immutable')
  - `levels`, `labels` and `names` are validated upon setting and are either copied or shallow-copied.
  - inplace setting of `levels` or `labels` now correctly invalidates the cached properties. (GH5238).
  - `__deepcopy__` now returns a shallow copy (currently: a view) of the data - allowing metadata changes.
  - `MultiIndex.astype()` now only allows `np.object_-like` dtypes and now returns a `MultiIndex` rather than an `Index`. (GH4039)
  - Added `is_` method to `Index` that allows fast equality comparison of views (similar to `np.may_share_memory` but no false positives, and changes on `levels` and `labels` setting on `MultiIndex`). (GH4859, GH4909)
  - Aliased `__iadd__` to `__add__`. (GH4996)
  - Added `is_` method to `Index` that allows fast equality comparison of views (similar to `np.may_share_memory` but no false positives, and changes on `levels` and `labels` setting on `MultiIndex`). (GH4859, GH4909)
- Infer and downcast dtype if `downcast='infer'` is passed to `fillna/ffill/bfill` (GH4604)
- `__nonzero__` for all `NDFrame` objects, will now raise a `ValueError`, this reverts back to (GH1073, GH4633) behavior. Add `.bool()` method to `NDFrame` objects to facilitate evaluating of single-element boolean Series
- `DataFrame.update()` no longer raises a `DataConflictError`, it now will raise a `ValueError` instead (if necessary) (GH4732)
- `Series.isin()` and `DataFrame.isin()` now raise a `TypeError` when passed a string (GH4763). Pass a list of one element (containing the string) instead.
- Remove undocumented/unused `kind` keyword argument from `read_excel`, and `ExcelFile`. (GH4713, GH4712)
- The method argument of `NDFrame.replace()` is valid again, so that a a list can be passed to `to_replace` (GH4743).



- provide automatic dtype conversions on `_reduce` operations (GH3371)
- exclude non-numerics if mixed types with datelike in `_reduce` operations (GH3371)
- default for `tupleize_cols` is now `False` for both `to_csv` and `read_csv`. Fair warning in 0.12 (GH3604)
- moved `timedeltas` support to `pandas.tseries.timedeltas.py`; add `timedeltas` string parsing, add top-level `to_timedelta` function
- `NDFrame` now is compatible with Python's `abs()` function (GH4821).
- raise a `TypeError` on invalid comparison ops on `Series/DataFrame` (e.g. `integer/datetime`) (GH4968)
- Added a new index type, `Float64Index`. This will be automatically created when passing floating values in index creation. This enables a pure label-based slicing paradigm that makes `[]`, `ix`, `loc` for scalar indexing and slicing work exactly the same. Indexing on other index types are preserved (and positional fallback for `[]`, `ix`), with the exception, that floating point slicing on indexes on non `Float64Index` will raise a `TypeError`, e.g. `Series(range(5))[3.5:4.5]` (GH263, issue:5375)
- Make Categorical repr nicer (GH4368)
- Remove deprecated `Factor` (GH3650)
- Remove deprecated `set_printoptions/reset_printoptions` (issue:3046)
- Remove deprecated `_verbose_info` (GH3215)
- Begin removing methods that don't make sense on `GroupBy` objects (GH4887).
- Remove deprecated `read_clipboard/to_clipboard/ExcelFile/ExcelWriter` from `pandas.io.parsers` (GH3717)
- All non-Index `NDFrames` (`Series`, `DataFrame`, `Panel`, `Panel4D`, `SparsePanel`, etc.), now support the entire set of arithmetic operators and arithmetic flex methods (`add`, `sub`, `mul`, etc.). `SparsePanel` does not support `pow` or `mod` with non-scalars. (GH3765)
- Arithmetic func factories are now passed real names (suitable for using with `super`) (GH5240)
- Provide NumPy compatibility with 1.7 for a calling convention like `np.prod(pandas_object)` as NumPy call with additional keyword args (GH4435)
- Provide `__dir__` method (and local context) for tab completion / remove `ipython` completers code (GH4501)
- Support non-unique axes in a `Panel` via indexing operations (GH4960)
- `.truncate` will raise a `ValueError` if invalid before and after dates are given (GH5242)
- `Timestamp` now supports `now/today/utcnow` class methods (GH5339)
- default for `display.max_seq_len` is now 100 rather than `None`. This activates truncated display ("...") of long sequences in various places. (GH3391)
- All division with `NDFrame` - likes is now `truedivision`, regardless of the `future` import. You can use `//` and `floordiv` to do integer division.

```
In [3]: arr = np.array([1, 2, 3, 4])
In [4]: arr2 = np.array([5, 3, 2, 1])
In [5]: arr / arr2
Out[5]: array([0, 0, 1, 4])

In [6]: pd.Series(arr) / pd.Series(arr2) # no future import required
Out[6]:
```

(continues on next page)

(continued from previous page)

```

0    0.200000
1    0.666667
2    1.500000
3    4.000000
dtype: float64

```

- `raise/warn SettingWithCopyError/Warning` exception/warning when setting of a copy thru chained assignment is detected, settable via option `mode.chained_assignment`
- test the list of NA values in the csv parser. add N/A, #NA as independent default na values ([GH5521](#))
- The refactoring involving “Series” deriving from `NDFrame` breaks `ipy2<=2.3.8`. an Issue has been opened against `ipy2` and a workaround is detailed in [GH5698](#). Thanks @JanSchulz.
- `Series.argmax` and `Series.argmin` are now aliased to `Series.idxmin` and `Series.idxmax`. These return the *index* of the min or max element respectively. Prior to 0.13.0 these would return the position of the min / max element ([GH6214](#))

### 38.24.5 Internal Refactoring

In 0.13.0 there is a major refactor primarily to subclass `Series` from `NDFrame`, which is the base class currently for `DataFrame` and `Panel`, to unify methods and behaviors. `Series` formerly subclassed directly from `ndarray`. ([GH4080](#), [GH3862](#), [GH816](#)) See *Internal Refactoring*

- Refactor of `series.py/frame.py/panel.py` to move common code to `generic.py`
- added `_setup_axes` to created generic `NDFrame` structures
- moved methods
  - `from_axes`, `_wrap_array`, `axes`, `ix`, `loc`, `iloc`, `shape`, `empty`, `swapaxes`, `transpose`, `pop`
  - `__iter__`, `keys`, `__contains__`, `__len__`, `__neg__`, `__invert__`
  - `convert_objects`, `as_blocks`, `as_matrix`, `values`
  - `__getstate__`, `__setstate__` (compat remains in `frame/panel`)
  - `__getattr__`, `__setattr__`
  - `_indexed_same`, `reindex_like`, `align`, `where`, `mask`
  - `fillna`, `replace` (`Series replace` is now consistent with `DataFrame`)
  - `filter` (also added `axis` argument to selectively filter on a different axis)
  - `reindex`, `reindex_axis`, `take`
  - `truncate` (moved to become part of `NDFrame`)
  - `isnull/notnull` now available on `NDFrame` objects
- These are API changes which make `Panel` more consistent with `DataFrame`
- `swapaxes` on a `Panel` with the same axes specified now return a copy
- support attribute access for setting
- `filter` supports same API as original `DataFrame filter`
- `fillna` refactored to `core/generic.py`, while `> 3ndim` is Not Implemented

- Series now inherits from `NDFrame` rather than directly from `ndarray`. There are several minor changes that affect the API.
- NumPy functions that do not support the array interface will now return `ndarrays` rather than series, e.g. `np.diff`, `np.ones_like`, `np.where`
- `Series(0.5)` would previously return the scalar `0.5`, this is no longer supported
- `TimeSeries` is now an alias for `Series`. the property `is_time_series` can be used to distinguish (if desired)
- Refactor of Sparse objects to use `BlockManager`
- Created a new block type in internals, `SparseBlock`, which can hold multi-dtypes and is non-consolidatable. `SparseSeries` and `SparseDataFrame` now inherit more methods from there hierarchy (`Series/DataFrame`), and no longer inherit from `SparseArray` (which instead is the object of the `SparseBlock`)
- Sparse suite now supports integration with non-sparse data. Non-float sparse data is supportable (partially implemented)
- Operations on sparse structures within `DataFrames` should preserve sparseness, merging type operations will convert to dense (and back to sparse), so might be somewhat inefficient
- enable `setitem` on `SparseSeries` for boolean/integer/slices
- `SparsePanels` implementation is unchanged (e.g. not using `BlockManager`, needs work)
- added `ftypes` method to `Series/DataFrame`, similar to `dtypes`, but indicates if the underlying is sparse/dense (as well as the dtype)
- All `NDFrame` objects now have a `_prop_attributes`, which can be used to indicate various values to propagate to a new object from an existing (e.g. name in `Series` will follow more automatically now)
- Internal type checking is now done via a suite of generated classes, allowing `isinstance(value, klass)` without having to directly import the class, courtesy of @jtratrner
- Bug in `Series` update where the parent frame is not updating its cache based on changes ([GH4080](#), [GH5216](#)) or types ([GH3217](#)), `fillna` ([GH3386](#))
- Indexing with dtype conversions fixed ([GH4463](#), [GH4204](#))
- Refactor `Series.reindex` to `core/generic.py` ([GH4604](#), [GH4618](#)), allow `method=` in reindexing on a `Series` to work
- `Series.copy` no longer accepts the `order` parameter and is now consistent with `NDFrame` copy
- Refactor `rename` methods to `core/generic.py`; fixes `Series.rename` for ([GH4605](#)), and adds `rename` with the same signature for `Panel`
- `Series` (for index) / `Panel` (for items) now as attribute access to its elements ([GH1903](#))
- Refactor `clip` methods to `core/generic.py` ([GH4798](#))
- Refactor of `_get_numeric_data/_get_bool_data` to `core/generic.py`, allowing `Series/Panel` functionality
- Refactor of `Series` arithmetic with time-like objects (`datetime/timedelta/time` etc.) into a separate, cleaned up wrapper class. ([GH4613](#))
- Complex compat for `Series` with `ndarray`. ([GH4819](#))
- Removed unnecessary `rwproperty` from codebase in favor of builtin `property`. ([GH4843](#))
- Refactor object level numeric methods (`mean/sum/min/max...`) from object level modules to `core/generic.py` ([GH4435](#)).

- Refactor cum objects to core/generic.py (GH4435), note that these have a more numpy-like function signature.
- `read_html()` now uses `TextParser` to parse HTML data from `bs4/lxml` (GH4770).
- Removed the `keep_internal` keyword parameter in `pandas/core/groupby.py` because it wasn't being used (GH5102).
- Base `DateOffsets` are no longer all instantiated on importing `pandas`, instead they are generated and cached on the fly. The internal representation and handling of `DateOffsets` has also been clarified. (GH5189, related GH5004)
- `MultiIndex` constructor now validates that passed levels and labels are compatible. (GH5213, GH5214)
- Unity `dropna` for `Series/DataFrame` signature (GH5250), tests from GH5234, courtesy of @rockg
- Rewrite `assert_almost_equal()` in cython for performance (GH4398)
- Added an internal `_update_inplace` method to facilitate updating `NDFrame` wrappers on inplace ops (only is for convenience of caller, doesn't actually prevent copies). (GH5247)

### 38.24.6 Bug Fixes

- `HDFStore`
  - raising an invalid `TypeError` rather than `ValueError` when appending with a different block ordering (GH4096)
  - `read_hdf` was not respecting as passed mode (GH4504)
  - appending a 0-len table will work correctly (GH4273)
  - `to_hdf` was raising when passing both arguments `append` and `table` (GH4584)
  - reading from a store with duplicate columns across dtypes would raise (GH4767)
  - Fixed a bug where `ValueError` wasn't correctly raised when column names weren't strings (GH4956)
  - A zero length series written in Fixed format not deserializing properly. (GH4708)
  - Fixed decoding perf issue on py3 (GH5441)
  - Validate levels in a multi-index before storing (GH5527)
  - Correctly handle `data_columns` with a `Panel` (GH5717)
- Fixed bug in `tslib.tz_convert(vals, tz1, tz2)`: it could raise `IndexError` exception while trying to access `trans[pos + 1]` (GH4496)
- The `by` argument now works correctly with the `layout` argument (GH4102, GH4014) in `*.hist` plotting methods
- Fixed bug in `PeriodIndex.map` where using `str` would return the `str` representation of the index (GH4136)
- Fixed test failure `test_time_series_plot_color_with_empty_kwargs` when using custom matplotlib default colors (GH4345)
- Fix running of stata IO tests. Now uses temporary files to write (GH4353)
- Fixed an issue where `DataFrame.sum` was slower than `DataFrame.mean` for integer valued frames (GH4365)
- `read_html` tests now work with Python 2.6 (GH4351)
- Fixed bug where network testing was throwing `NameError` because a local variable was undefined (GH4381)

- In `to_json`, raise if a passed `orient` would cause loss of data because of a duplicate index (GH4359)
- In `to_json`, fix date handling so milliseconds are the default timestamp as the docstring says (GH4362).
- `as_index` is no longer ignored when doing `groupby` apply (GH4648, GH3417)
- JSON NaT handling fixed, NaTs are now serialized to `null` (GH4498)
- Fixed JSON handling of escapable characters in JSON object keys (GH4593)
- Fixed passing `keep_default_na=False` when `na_values=None` (GH4318)
- Fixed bug with `values` raising an error on a DataFrame with duplicate columns and mixed dtypes, surfaced in (GH4377)
- Fixed bug with duplicate columns and type conversion in `read_json` when `orient='split'` (GH4377)
- Fixed JSON bug where locales with decimal separators other than `'.'` threw exceptions when encoding / decoding certain values. (GH4918)
- Fix `.iat` indexing with a `PeriodIndex` (GH4390)
- Fixed an issue where `PeriodIndex` joining with self was returning a new instance rather than the same instance (GH4379); also adds a test for this for the other index types
- Fixed a bug with all the dtypes being converted to object when using the CSV cparser with the `usecols` parameter (GH3192)
- Fix an issue in merging blocks where the resulting DataFrame had partially set `_ref_locs` (GH4403)
- Fixed an issue where hist subplots were being overwritten when they were called using the top level matplotlib API (GH4408)
- Fixed a bug where calling `Series.astype(str)` would truncate the string (GH4405, GH4437)
- Fixed a py3 compat issue where bytes were being repr'd as tuples (GH4455)
- Fixed Panel attribute naming conflict if item is named `'a'` (GH3440)
- Fixed an issue where duplicate indexes were raising when plotting (GH4486)
- Fixed an issue where `cumsum` and `cumprod` didn't work with bool dtypes (GH4170, GH4440)
- Fixed Panel slicing issued in `xs` that was returning an incorrect dimmed object (GH4016)
- Fix resampling bug where custom reduce function not used if only one group (GH3849, GH4494)
- Fixed Panel assignment with a transposed frame (GH3830)
- Raise on set indexing with a Panel and a Panel as a value which needs alignment (GH3777)
- `frozenset` objects now raise in the `Series` constructor (GH4482, GH4480)
- Fixed issue with sorting a duplicate multi-index that has multiple dtypes (GH4516)
- Fixed bug in `DataFrame.set_values` which was causing name attributes to be lost when expanding the index. (GH3742, GH4039)
- Fixed issue where individual names, levels and labels could be set on `MultiIndex` without validation (GH3714, GH4039)
- Fixed (GH3334) in `pivot_table`. Margins did not compute if values is the index.
- Fix bug in having a rhs of `np.timedelta64` or `np.offsets.DateOffset` when operating with date-times (GH4532)
- Fix arithmetic with series/datetimeindex and `np.timedelta64` not working the same (GH4134) and buggy `timedelta` in NumPy 1.6 (GH4135)

- Fix bug in `pd.read_clipboard` on windows with PY3 ([GH4561](#)); not decoding properly
- `tslib.get_period_field()` and `tslib.get_period_field_arr()` now raise if code argument out of range ([GH4519](#), [GH4520](#))
- Fix boolean indexing on an empty series loses index names ([GH4235](#)), `infer_dtype` works with empty arrays.
- Fix reindexing with multiple axes; if an axes match was not replacing the current axes, leading to a possible lazy frequency inference issue ([GH3317](#))
- Fixed issue where `DataFrame.apply` was reraising exceptions incorrectly (causing the original stack trace to be truncated).
- Fix selection with `ix/loc` and `non_unique` selectors ([GH4619](#))
- Fix assignment with `iloc/loc` involving a dtype change in an existing column ([GH4312](#), [GH5702](#)) have internal `setitem_with_indexer` in `core/indexing` to use `Block.setitem`
- Fixed bug where thousands operator was not handled correctly for floating point numbers in `csv_import` ([GH4322](#))
- Fix an issue with `CacheableOffset` not properly being used by many `DateOffset`; this prevented the `DateOffset` from being cached ([GH4609](#))
- Fix boolean comparison with a `DataFrame` on the lhs, and a list/tuple on the rhs ([GH4576](#))
- Fix error/dtype conversion with `setitem` of `None` on `Series/DataFrame` ([GH4667](#))
- Fix decoding based on a passed in non-default encoding in `pd.read_stata` ([GH4626](#))
- Fix `DataFrame.from_records` with a plain-vanilla `ndarray`. ([GH4727](#))
- Fix some inconsistencies with `Index.rename` and `MultiIndex.rename`, etc. ([GH4718](#), [GH4628](#))
- Bug in using `iloc/loc` with a cross-sectional and duplicate indices ([GH4726](#))
- Bug with using `QUOTE_NONE` with `to_csv` causing `Exception`. ([GH4328](#))
- Bug with `Series` indexing not raising an error when the right-hand-side has an incorrect length ([GH2702](#))
- Bug in multi-indexing with a partial string selection as one part of a `MultiIndex` ([GH4758](#))
- Bug with reindexing on the index with a non-unique index will now raise `ValueError` ([GH4746](#))
- Bug in setting with `loc/ix` a single indexer with a multi-index axis and a NumPy array, related to ([GH3777](#))
- Bug in concatenation with duplicate columns across dtypes not merging with `axis=0` ([GH4771](#), [GH4975](#))
- Bug in `iloc` with a slice index failing ([GH4771](#))
- Incorrect error message with no colspecs or width in `read_fwf`. ([GH4774](#))
- Fix bugs in indexing in a `Series` with a duplicate index ([GH4548](#), [GH4550](#))
- Fixed bug with reading compressed files with `read_fwf` in Python 3. ([GH3963](#))
- Fixed an issue with a duplicate index and assignment with a dtype change ([GH4686](#))
- Fixed bug with reading compressed files in as `bytes` rather than `str` in Python 3. Simplifies bytes-producing file-handling in Python 3 ([GH3963](#), [GH4785](#)).
- Fixed an issue related to `ticklocs/ticklabels` with log scale bar plots across different versions of `matplotlib` ([GH4789](#))
- Suppressed `DeprecationWarning` associated with internal calls issued by `repr()` ([GH4391](#))
- Fixed an issue with a duplicate index and duplicate selector with `.loc` ([GH4825](#))

- Fixed an issue with `DataFrame.sort_index` where, when sorting by a single column and passing a list for ascending, the argument for ascending was being interpreted as `True` (GH4839, GH4846)
- Fixed `Panel.tshift` not working. Added *freq* support to `Panel.shift` (GH4853)
- Fix an issue in `TextFileReader` w/ Python engine (i.e. `PythonParser`) with thousands != “,” (GH4596)
- Bug in `getitem` with a duplicate index when using `where` (GH4879)
- Fix Type inference code coerces float column into datetime (GH4601)
- Fixed `_ensure_numeric` does not check for complex numbers (GH4902)
- Fixed a bug in `Series.hist` where two figures were being created when the `by` argument was passed (GH4112, GH4113).
- Fixed a bug in `convert_objects` for > 2 ndims (GH4937)
- Fixed a bug in `DataFrame/Panel` cache insertion and subsequent indexing (GH4939, GH5424)
- Fixed string methods for `FrozenNDArray` and `FrozenList` (GH4929)
- Fixed a bug with setting invalid or out-of-range values in indexing enlargement scenarios (GH4940)
- Tests for `fillna` on empty `Series` (GH4346), thanks @immerrr
- Fixed `copy()` to shallow copy axes/indices as well and thereby keep separate metadata. (GH4202, GH4830)
- Fixed `skiprows` option in Python parser for `read_csv` (GH4382)
- Fixed bug preventing `cut` from working with `np.inf` levels without explicitly passing labels (GH3415)
- Fixed wrong check for overlapping in `DatetimeIndex.union` (GH4564)
- Fixed conflict between thousands separator and date parser in `csv_parser` (GH4678)
- Fix appending when dtypes are not the same (error showing mixing float/np.datetime64) (GH4993)
- Fix repr for `DateOffset`. No longer show duplicate entries in `kwds`. Removed unused offset fields. (GH4638)
- Fixed wrong index name during `read_csv` if using `usecols`. Applies to `c` parser only. (GH4201)
- `Timestamp` objects can now appear in the left hand side of a comparison operation with a `Series` or `DataFrame` object (GH4982).
- Fix a bug when indexing with `np.nan` via `iloc/loc` (GH5016)
- Fixed a bug where low memory `c` parser could create different types in different chunks of the same file. Now coerces to numerical type or raises warning. (GH3866)
- Fix a bug where reshaping a `Series` to its own shape raised `TypeError` (GH4554) and other reshaping issues.
- Bug in setting with `ix/loc` and a mixed int/string index (GH4544)
- Make sure series-series boolean comparisons are label based (GH4947)
- Bug in multi-level indexing with a `Timestamp` partial indexer (GH4294)
- Tests/fix for multi-index construction of an all-nan frame (GH4078)
- Fixed a bug where `read_html()` wasn't correctly inferring values of tables with commas (GH5029)
- Fixed a bug where `read_html()` wasn't providing a stable ordering of returned tables (GH4770, GH5029).
- Fixed a bug where `read_html()` was incorrectly parsing when passed `index_col=0` (GH5066).
- Fixed a bug where `read_html()` was incorrectly inferring the type of headers (GH5048).
- Fixed a bug where `DatetimeIndex` joins with `PeriodIndex` caused a stack overflow (GH3899).



- Fixed a bug where `groupby` objects didn't allow plots (GH5102).
- Fixed a bug where `groupby` objects weren't tab-completing column names (GH5102).
- Fixed a bug where `groupby.plot()` and `friends` were duplicating figures multiple times (GH5102).
- Provide automatic conversion of `object` dtypes on `fillna`, related (GH5103)
- Fixed a bug where default options were being overwritten in the option parser cleaning (GH5121).
- Treat a list/ndarray identically for `iloc` indexing with list-like (GH5006)
- Fix `MultiIndex.get_level_values()` with missing values (GH5074)
- Fix bound checking for `Timestamp()` with `datetime64` input (GH4065)
- Fix a bug where `TestReadHtml` wasn't calling the correct `read_html()` function (GH5150).
- Fix a bug with `NDFrame.replace()` which made replacement appear as though it was (incorrectly) using regular expressions (GH5143).
- Fix better error message for `to_datetime` (GH4928)
- Made sure different locales are tested on `travis-ci` (GH4918). Also adds a couple of utilities for getting locales and setting locales with a context manager.
- Fixed segfault on `isnull(MultiIndex)` (now raises an error instead) (GH5123, GH5125)
- Allow duplicate indices when performing operations that align (GH5185, GH5639)
- Compound dtypes in a constructor raise `NotImplementedError` (GH5191)
- Bug in comparing duplicate frames (GH4421) related
- Bug in `describe` on duplicate frames
- Bug in `to_datetime` with a format and `coerce=True` not raising (GH5195)
- Bug in `loc` setting with multiple indexers and a rhs of a Series that needs broadcasting (GH5206)
- Fixed bug where inplace setting of levels or labels on `MultiIndex` would not clear cached values property and therefore return wrong values. (GH5215)
- Fixed bug where filtering a grouped `DataFrame` or `Series` did not maintain the original ordering (GH4621).
- Fixed `Period` with a business date freq to always roll-forward if on a non-business date. (GH5203)
- Fixed bug in Excel writers where frames with duplicate column names weren't written correctly. (GH5235)
- Fixed issue with `drop` and a non-unique index on `Series` (GH5248)
- Fixed seg fault in C parser caused by passing more names than columns in the file. (GH5156)
- Fix `Series.isin` with date/time-like dtypes (GH5021)
- C and Python Parser can now handle the more common multi-index column format which doesn't have a row for index names (GH4702)
- Bug when trying to use an out-of-bounds date as an object dtype (GH5312)
- Bug when trying to display an embedded `PandasObject` (GH5324)
- Allows operating of `Timestamps` to return a `datetime` if the result is out-of-bounds related (GH5312)
- Fix return value/type signature of `initObjToJSON()` to be compatible with `numpy's import_array()` (GH5334, GH5326)
- Bug when renaming then `set_index` on a `DataFrame` (GH5344)



- Test suite no longer leaves around temporary files when testing graphics. (GH5347) (thanks for catching this @yarikoptic!)
- Fixed html tests on win32. (GH4580)
- Make sure that head/tail are iloc based, (GH5370)
- Fixed bug for PeriodIndex string representation if there are 1 or 2 elements. (GH5372)
- The GroupBy methods transform and filter can be used on Series and DataFrames that have repeated (non-unique) indices. (GH4620)
- Fix empty series not printing name in repr (GH4651)
- Make tests create temp files in temp directory by default. (GH5419)
- pd.to\_timedelta of a scalar returns a scalar (GH5410)
- pd.to\_timedelta accepts NaN and NaT, returning NaT instead of raising (GH5437)
- performance improvements in isnull on larger size pandas objects
- Fixed various setitem with 1d ndarray that does not have a matching length to the indexer (GH5508)
- Bug in getitem with a multi-index and iloc (GH5528)
- Bug in delitem on a Series (GH5542)
- Bug fix in apply when using custom function and objects are not mutated (GH5545)
- Bug in selecting from a non-unique index with loc (GH5553)
- Bug in groupby returning non-consistent types when user function returns a None, (GH5592)
- Work around regression in numpy 1.7.0 which erroneously raises IndexError from ndarray.item (GH5666)
- Bug in repeated indexing of object with resultant non-unique index (GH5678)
- Bug in fillna with Series and a passed series/dict (GH5703)
- Bug in groupby transform with a datetime-like grouper (GH5712)
- Bug in multi-index selection in PY3 when using certain keys (GH5725)
- Row-wise concat of differing dtypes failing in certain cases (GH5754)

## 38.25 pandas 0.12.0

Release date: 2013-07-24

### 38.25.1 New Features

- pd.read\_html() can now parse HTML strings, files or urls and returns a list of DataFrame s courtesy of @cpcloud. (GH3477, GH3605, GH3606)
- Support for reading Amazon S3 files. (GH3504)
- Added module for reading and writing JSON strings/files: pandas.io.json includes to\_json DataFrame/Series method, and a read\_json top-level reader various issues (GH1226, GH3804, GH3876, GH3867, GH1305)
- Added module for reading and writing Stata files: pandas.io.stata (GH1512) includes to\_stata DataFrame method, and a read\_stata top-level reader

- Added support for writing in `to_csv` and reading in `read_csv`, multi-index columns. The `header` option in `read_csv` now accepts a list of the rows from which to read the index. Added the option, `tupleize_cols` to provide compatibility for the pre 0.12 behavior of writing and reading multi-index columns via a list of tuples. The default in 0.12 is to write lists of tuples and *not* interpret list of tuples as a multi-index column. Note: The default value will change in 0.12 to make the default *to* write and read multi-index columns in the new format. ([GH3571](#), [GH1651](#), [GH3141](#))
- Add iterator to `Series.str` ([GH3638](#))
- `pd.set_option()` now allows N option, value pairs ([GH3667](#)).
- Added keyword parameters for different types of `scatter_matrix` subplots
- A `filter` method on grouped Series or DataFrames returns a subset of the original ([GH3680](#), [GH919](#))
- Access to historical Google Finance data in `pandas.io.data` ([GH3814](#))
- DataFrame plotting methods can sample column colors from a Matplotlib colormap via the `colormap` keyword. ([GH3860](#))

### 38.25.2 Improvements to existing features

- Fixed various issues with internal pprinting code, the `repr()` for various objects including `TimeStamp` and `Index` now produces valid Python code strings and can be used to recreate the object, ([GH3038](#), [GH3379](#), [GH3251](#), [GH3460](#))
- `convert_objects` now accepts a `copy` parameter (defaults to `True`)
- `HDFStore`
  - will retain index attributes (`freq,tz,name`) on recreation ([GH3499](#),:issue:4098)
  - will warn with a `AttributeConflictWarning` if you are attempting to append an index with a different frequency than the existing, or attempting to append an index with a different name than the existing
  - support datelike columns with a timezone as `data_columns` ([GH2852](#))
  - table writing performance improvements.
  - support python3 (via `PyTables 3.0.0`) ([GH3750](#))
- Add modulo operator to Series, DataFrame
- Add `date` method to `DatetimeIndex`
- Add `dropna` argument to `pivot_table` (:issue: 3820)
- Simplified the API and added a `describe` method to Categorical
- `melt` now accepts the optional parameters `var_name` and `value_name` to specify custom column names of the returned DataFrame ([GH3649](#)), thanks @hoechenberger. If `var_name` is not specified and `dataframe.columns.name` is not `None`, then this will be used as the `var_name` ([GH4144](#)). Also support for `MultiIndex` columns.
- clipboard functions use `pyperclip` (no dependencies on Windows, alternative dependencies offered for Linux) ([GH3837](#)).
- Plotting functions now raise a `TypeError` before trying to plot anything if the associated objects have a dtype of `object` ([GH1818](#), [GH3572](#), [GH3911](#), [GH3912](#)), but they will try to convert object arrays to numeric arrays if possible so that you can still plot, for example, an object array with floats. This happens before any drawing takes place which eliminates any spurious plots from showing up.
- Added `Faq` section on `repr` display options, to help users customize their setup.

- where operations that result in block splitting are much faster ([GH3733](#))
- Series and DataFrame hist methods now take a `figsize` argument ([GH3834](#))
- DatetimeIndexes no longer try to convert mixed-integer indexes during join operations ([GH3877](#))
- Add `unit` keyword to `Timestamp` and `to_datetime` to enable passing of integers or floats that are in an epoch unit of `D`, `s`, `ms`, `us`, `ns`, thanks @mtkini ([GH3969](#)) (e.g. unix timestamps or epoch `s`, with fractional seconds allowed) ([GH3540](#))
- DataFrame corr method (`spearman`) is now cythonized.
- Improved network test decorator to catch `IOError` (and therefore `URLError` as well). Added `with_connectivity_check` decorator to allow explicitly checking a website as a proxy for seeing if there is network connectivity. Plus, new `optional_args` decorator factory for decorators. ([GH3910](#), [GH3914](#))
- `read_csv` will now throw a more informative error message when a file contains no columns, e.g., all newline characters
- Added `layout` keyword to `DataFrame.hist()` for more customizable layout ([GH4050](#))
- `Timestamp.min` and `Timestamp.max` now represent valid `Timestamp` instances instead of the default `datetime.min` and `datetime.max` (respectively), thanks @SleepingPills
- `read_html` now raises when no tables are found and `BeautifulSoup==4.2.0` is detected ([GH4214](#))

### 38.25.3 API Changes

- `HDFStore`
  - When removing an object, `remove(key)` raises `KeyError` if the key is not a valid store object.
  - raise a `TypeError` on passing `where` or `columns` to select with a `Storer`; these are invalid parameters at this time ([GH4189](#))
  - can now specify an `encoding` option to `append/put` to enable alternate encodings ([GH3750](#))
  - enable support for `iterator/chunksize` with `read_hdf`
- The `repr()` for (Multi)Index now obeys `display.max_seq_items` rather than NumPy threshold print options. ([GH3426](#), [GH3466](#))
- Added `mangle_dupe_cols` option to `read_table/csv`, allowing users to control legacy behaviour re dupe cols (A, A.1, A.2 vs A, A) ([GH3468](#)) Note: The default value will change in 0.12 to the “no mangle” behaviour, If your code relies on this behaviour, explicitly specify `mangle_dupe_cols=True` in your calls.
- Do not allow astypes on `datetime64[ns]` except to `object`, and `timedelta64[ns]` to `object/int` ([GH3425](#))
- The behavior of `datetime64` dtypes has changed with respect to certain so-called reduction operations ([GH3726](#)). The following operations now raise a `TypeError` when performed on a `Series` and return an *empty* `Series` when performed on a `DataFrame` similar to performing these operations on, for example, a `DataFrame` of `slice` objects: - `sum`, `prod`, `mean`, `std`, `var`, `skew`, `kurt`, `corr`, and `cov`
- Do not allow `datetimelike/timedeltalike` creation except with valid types (e.g. cannot pass `datetime64[ms]`) ([GH3423](#))
- Add `squeeze` keyword to `groupby` to allow reduction from `DataFrame` -> `Series` if groups are unique. Regression from 0.10.1, partial revert on ([GH2893](#)) with ([GH3596](#))
- Raise on `iloc` when boolean indexing with a label based indexer mask e.g. a boolean `Series`, even with integer labels, will raise. Since `iloc` is purely positional based, the labels on the `Series` are not alignable ([GH3631](#))

- The `raise_on_error` option to plotting methods is obviated by [GH3572](#), so it is removed. Plots now always raise when data cannot be plotted or the object being plotted has a dtype of object.
- `DataFrame.interpolate()` is now deprecated. Please use `DataFrame.fillna()` and `DataFrame.replace()` instead ([GH3582](#), [GH3675](#), [GH3676](#)).
- the method and axis arguments of `DataFrame.replace()` are deprecated
- `DataFrame.replace` 's `infer_types` parameter is removed and now performs conversion by default. ([GH3907](#))
- Deprecate `display.height`, `display.width` is now only a formatting option does not control triggering of summary, similar to < 0.11.0.
- Add the keyword `allow_duplicates` to `DataFrame.insert` to allow a duplicate column to be inserted if `True`, default is `False` (same as prior to 0.12) ([GH3679](#))
- io API changes
  - added `pandas.io.api` for i/o imports
  - removed Excel support to `pandas.io.excel`
  - added top-level `pd.read_sql` and `to_sql` `DataFrame` methods
  - removed clipboard support to `pandas.io.clipboard`
  - replace top-level and instance methods `save` and `load` with top-level `read_pickle` and `to_pickle` instance method, `save` and `load` will give deprecation warning.
- the method and axis arguments of `DataFrame.replace()` are deprecated
- set `FutureWarning` to require `data_source`, and to replace year/month with expiry date in `pandas.io` options. This is in preparation to add options data from Google ([GH3822](#))
- the method and axis arguments of `DataFrame.replace()` are deprecated
- Implement `__nonzero__` for `NDFrame` objects ([GH3691](#), [GH3696](#))
- `as_matrix` with mixed signed and unsigned dtypes will result in 2 x the lcd of the unsigned as an int, maxing with `int64`, to avoid precision issues ([GH3733](#))
- `na_values` in a list provided to `read_csv/read_excel` will match string and numeric versions e.g. `na_values=['99']` will match 99 whether the column ends up being int, float, or string ([GH3611](#))
- `read_html` now defaults to `None` when reading, and falls back on `bs4 + html5lib` when `lxml` fails to parse. a list of parsers to try until success is also valid
- more consistency in the `to_datetime` return types (give string/array of string inputs) ([GH3888](#))
- The internal pandas class hierarchy has changed (slightly). The previous `PandasObject` now is called `PandasContainer` and a new `PandasObject` has become the baseclass for `PandasContainer` as well as `Index`, `Categorical`, `GroupBy`, `SparseList`, and `SparseArray` (+ their base classes). Currently, `PandasObject` provides string methods (from `StringMixin`). ([GH4090](#), [GH4092](#))
- New `StringMixin` that, given a `__unicode__` method, gets Python 2 and Python 3 compatible string methods (`__str__`, `__bytes__`, and `__repr__`). Plus string safety throughout. Now employed in many places throughout the pandas library. ([GH4090](#), [GH4092](#))

### 38.25.4 Experimental Features

- Added experimental `CustomBusinessDay` class to support `DateOffsets` with custom holiday calendars and custom weekmasks. ([GH2301](#))

### 38.25.5 Bug Fixes

- Fixed an esoteric excel reading bug, xlrd>= 0.9.0 now required for excel support. Should provide python3 support (for reading) which has been lacking. (GH3164)
- Disallow Series constructor called with MultiIndex which caused segfault (GH4187)
- Allow unioning of date ranges sharing a timezone (GH3491)
- Fix to\_csv issue when having a large number of rows and NaT in some columns (GH3437)
- `.loc` was not raising when passed an integer list (GH3449)
- Unordered time series selection was misbehaving when using label slicing (GH3448)
- Fix sorting in a frame with a list of columns which contains datetime64[ns] dtypes (GH3461)
- DataFrames fetched via FRED now handle '.' as a NaN. (GH3469)
- Fix regression in a DataFrame apply with axis=1, objects were not being converted back to base dtypes correctly (GH3480)
- Fix issue when storing uint dtypes in an HDFStore. (GH3493)
- Non-unique index support clarified (GH3468)
  - Addressed handling of dupe columns in df.to\_csv new and old (GH3454, GH3457)
  - Fix assigning a new index to a duplicate index in a DataFrame would fail (GH3468)
  - Fix construction of a DataFrame with a duplicate index
  - ref\_locs support to allow duplicative indices across dtypes, allows iget support to always find the index (even across dtypes) (GH2194)
  - applymap on a DataFrame with a non-unique index now works (removed warning) (GH2786), and fix (GH3230)
  - Fix to\_csv to handle non-unique columns (GH3495)
  - Duplicate indexes with getitem will return items in the correct order (GH3455, GH3457) and handle missing elements like unique indices (GH3561)
  - Duplicate indexes with and empty DataFrame.from\_records will return a correct frame (GH3562)
  - Concat to produce a non-unique columns when duplicates are across dtypes is fixed (GH3602)
  - Non-unique indexing with a slice via loc and friends fixed (GH3659)
  - Allow insert/delete to non-unique columns (GH3679)
  - Extend reindex to correctly deal with non-unique indices (GH3679)
  - DataFrame.itertuples() now works with frames with duplicate column names (GH3873)
  - Bug in non-unique indexing via iloc (GH4017); added takeable argument to reindex for location-based taking
  - Allow non-unique indexing in series via .ix/.loc and \_\_getitem\_\_ (GH4246)
  - Fixed non-unique indexing memory allocation issue with .ix/.loc (GH4280)
- Fixed bug in groupby with empty series referencing a variable before assignment. (GH3510)
- Allow index name to be used in groupby for non MultiIndex (GH4014)
- Fixed bug in mixed-frame assignment with aligned series (GH3492)

- Fixed bug in selecting month/quarter/year from a series would not select the time element on the last day (GH3546)
- Fixed a couple of MultiIndex rendering bugs in `df.to_html()` (GH3547, GH3553)
- Properly convert `np.datetime64` objects in a Series (GH3416)
- Raise a `TypeError` on invalid datetime/timedelta operations e.g. add datetimes, multiple timedelta x datetime
- Fix `.diff` on datelike and timedelta operations (GH3100)
- `combine_first` not returning the same dtype in cases where it can (GH3552)
- Fixed bug with `Panel.transpose` argument aliases (GH3556)
- Fixed platform bug in `PeriodIndex.take` (GH3579)
- Fixed bud in incorrect conversion of `datetime64[ns]` in `combine_first` (GH3593)
- Fixed bug in `reset_index` with `NaN` in a multi-index (GH3586)
- `fillna` methods now raise a `TypeError` when the `value` parameter is a list or tuple.
- Fixed bug where a time-series was being selected in preference to an actual column name in a frame (GH3594)
- Make `secondary_y` work properly for bar plots (GH3598)
- Fix modulo and integer division on Series/DataFrames to act similarly to `float` dtypes to return `np.nan` or `np.inf` as appropriate (GH3590)
- Fix incorrect dtype on `groupby` with `as_index=False` (GH3610)
- Fix `read_csv/read_excel` to correctly encode identical `na_values`, e.g. `na_values=[-999.0, -999]` was failing (GH3611)
- Disable HTML output in `qtconsole` again. (GH3657)
- Reworked the new repr display logic, which users found confusing. (GH3663)
- Fix indexing issue in `ndim >= 3` with `iloc` (GH3617)
- Correctly parse date columns with embedded (nan/NaT) into `datetime64[ns]` dtype in `read_csv` when `parse_dates` is specified (GH3062)
- Fix not consolidating before `to_csv` (GH3624)
- Fix alignment issue when `setitem` in a DataFrame with a piece of a DataFrame (GH3626) or a mixed DataFrame and a Series (GH3668)
- Fix plotting of unordered `DatetimeIndex` (GH3601)
- `sql.write_frame` failing when writing a single column to `sqlite` (GH3628), thanks to @stonebig
- Fix pivoting with `nan` in the index (GH3558)
- Fix running of `bs4` tests when it is not installed (GH3605)
- Fix parsing of html table (GH3606)
- `read_html()` now only allows a single backend: `html5lib` (GH3616)
- `convert_objects` with `convert_dates='coerce'` was parsing some single-letter strings into today's date
- `DataFrame.from_records` did not accept empty recarrays (GH3682)
- `DataFrame.to_csv` will succeed with the deprecated option `nanRep`, @tdsmith
- `DataFrame.to_html` and `DataFrame.to_latex` now accept a path for their first argument (GH3702)

- Fix file tokenization error with r delimiter and quoted fields ([GH3453](#))
- Groupby transform with item-by-item not upcasting correctly ([GH3740](#))
- Incorrectly read a HDFStore multi-index Frame with a column specification ([GH3748](#))
- `read_html` now correctly skips tests ([GH3741](#))
- PandasObjects raise `TypeError` when trying to hash ([GH3882](#))
- Fix incorrect arguments passed to `concat` that are not list-like (e.g. `concat(df1,df2)`) ([GH3481](#))
- Correctly parse when passed the `dtype=str` (or other variable-len string dtypes) in `read_csv` ([GH3795](#))
- Fix index name not propagating when using `loc/ix` ([GH3880](#))
- Fix groupby when applying a custom function resulting in a returned DataFrame was not converting dtypes ([GH3911](#))
- Fixed a bug where `DataFrame.replace` with a compiled regular expression in the `to_replace` argument wasn't working ([GH3907](#))
- Fixed `__truediv__` in Python 2.7 with `numexpr` installed to actually do true division when dividing two integer arrays with at least 10000 cells total ([GH3764](#))
- Indexing with a string with seconds resolution not selecting from a time index ([GH3925](#))
- csv parsers would loop infinitely if `iterator=True` but no `chunksize` was specified ([GH3967](#)), Python parser failing with `chunksize=1`
- Fix index name not propagating when using `shift`
- Fixed `dropna=False` being ignored with multi-index stack ([GH3997](#))
- Fixed flattening of columns when renaming MultiIndex columns DataFrame ([GH4004](#))
- Fix `Series.clip` for datetime series. NA/NaN threshold values will now throw `ValueError` ([GH3996](#))
- Fixed insertion issue into DataFrame, after rename ([GH4032](#))
- Fixed testing issue where too many sockets were open thus leading to a connection reset issue ([GH3982](#), [GH3985](#), [GH4028](#), [GH4054](#))
- Fixed failing tests in `test_yahoo`, `test_google` where symbols were not retrieved but were being accessed ([GH3982](#), [GH3985](#), [GH4028](#), [GH4054](#))
- `Series.hist` will now take the figure from the current environment if one is not passed
- Fixed bug where a 1xN DataFrame would barf on a 1xN mask ([GH4071](#))
- Fixed running of `tox` under python3 where the pickle import was getting rewritten in an incompatible way ([GH4062](#), [GH4063](#))
- Fixed bug where `sharex` and `sharey` were not being passed to `grouped_hist` ([GH4089](#))
- Fix bug where `HDFStore` will fail to append because of a different block ordering on-disk ([GH4096](#))
- Better error messages on inserting incompatible columns to a frame ([GH4107](#))
- Fixed bug in `DataFrame.replace` where a nested dict wasn't being iterated over when `regex=False` ([GH4115](#))
- Fixed bug in `convert_objects(convert_numeric=True)` where a mixed numeric and object Series/Frame was not converting properly ([GH4119](#))
- Fixed bugs in multi-index selection with column multi-index and duplicates ([GH4145](#), [GH4146](#))
- Fixed bug in the parsing of microseconds when using the `format` argument in `to_datetime` ([GH4152](#))



- Fixed bug in `PandasAutoDateLocator` where `invert_xaxis` triggered incorrectly `MilliSecondLocator` (GH3990)
- Fixed bug in `Series.where` where broadcasting a single element input vector to the length of the series resulted in multiplying the value inside the input (GH4192)
- Fixed bug in plotting that wasn't raising on invalid colormap for matplotlib 1.1.1 (GH4215)
- Fixed the legend displaying in `DataFrame.plot(kind='kde')` (GH4216)
- Fixed bug where Index slices weren't carrying the name attribute (GH4226)
- Fixed bug in initializing `DatetimeIndex` with an array of strings in a certain time zone (GH4229)
- Fixed bug where `html5lib` wasn't being properly skipped (GH4265)
- Fixed bug where `get_data_famafrench` wasn't using the correct file edges (GH4281)

## 38.26 pandas 0.11.0

**Release date:** 2013-04-22

### 38.26.1 New Features

- New documentation section, `10 Minutes to Pandas`
- New documentation section, `Cookbook`
- Allow mixed dtypes (e.g `float32/float64/int32/int16/int8`) to coexist in `DataFrames` and propagate in operations
- Add function to `pandas.io.data` for retrieving stock index components from Yahoo! finance (GH2795)
- Support slicing with time objects (GH2681)
- Added `.iloc` attribute, to support strict integer based indexing, analogous to `.ix` (GH2922)
- Added `.loc` attribute, to support strict label based indexing, analogous to `.ix` (GH3053)
- Added `.iat` attribute, to support fast scalar access via integers (replaces `iget_value/iset_value`)
- Added `.at` attribute, to support fast scalar access via labels (replaces `get_value/set_value`)
- Moved functionality from `irow, icol, iget_value/iset_value` to `.iloc` indexer (via `_ixs` methods in each object)
- Added support for expression evaluation using the `numexpr` library
- Added `convert=boolean` to take routines to translate negative indices to positive, defaults to `True`
- Added `to_series()` method to indices, to facilitate the creation of indexers (GH3275)

### 38.26.2 Improvements to existing features

- Improved performance of `df.to_csv()` by up to 10x in some cases. (GH3059)
- added `blocks` attribute to `DataFrames`, to return a dict of dtypes to homogeneously dtyped `DataFrames`
- added keyword `convert_numeric` to `convert_objects()` to try to convert object dtypes to numeric types (default is `False`)



```
In [2]: p
Out[2]:
<class 'pandas.core.panel.Panel'>
Dimensions: 3 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: ItemA to ItemC
Major_axis axis: 2001-01-02 00:00:00 to 2001-01-05 00:00:00
Minor_axis axis: A to D
```

	A	B	C	D
2001-01-02	0.469112	-0.282863	-1.509059	-1.135632
2001-01-03	1.212112	-0.173215	0.119209	-1.044236
2001-01-04	-0.861849	-2.104569	-0.494929	1.071804
2001-01-05	0.721555	-0.706771	-1.039575	0.271860

[illegible]

- Improvement to Yahoo API access in `pd.io.data.Options` (GH2758)

- added option `display.max_seq_items` to control the number of elements printed per sequence pprinting it. (GH2979)
- added option `display.chop_threshold` to control display of small numerical values. (GH2739)
- added option `display.max_info_rows` to prevent verbose\_info from being calculated for frames above 1M rows (configurable). (GH2807, GH2918)
- `value_counts()` now accepts a “normalize” argument, for normalized histograms. (GH2710).
- `DataFrame.from_records` now accepts not only dicts but any instance of the collections.Mapping ABC.
- Allow selection semantics via a string with a datelike index to work in both Series and DataFrames (GH3070)

```
In [5]: idx = pd.date_range("2001-10-1", periods=5, freq='M')
In [6]: ts = pd.Series(np.random.rand(len(idx)), index=idx)
In [7]: ts['2001']
Out[7]:
2001-10-31    0.838796
2001-11-30    0.897333
2001-12-31    0.732592
Freq: M, dtype: float64

In [8]: df = pd.DataFrame(dict(A = ts))
In [9]: df['2001']
Out[9]:
           A
2001-10-31  0.838796
2001-11-30  0.897333
2001-12-31  0.732592
```

- added option `display.mpl_style` providing a sleeker visual style for plots. Based on <https://gist.github.com/huyng/816622> (GH3075).
- Improved performance across several core functions by taking memory ordering of arrays into account. Courtesy of @stephenwlin (GH3130)
- Improved performance of groupby transform method (GH2121)
- Handle “ragged” CSV files missing trailing delimiters in rows with missing fields when also providing explicit list of column names (so the parser knows how many columns to expect in the result) (GH2981)
- On a mixed DataFrame, allow setting with indexers with ndarray/DataFrame on rhs (GH3216)
- Treat boolean values as integers (values 1 and 0) for numeric operations. (GH2641)
- Add `time` method to DatetimeIndex (GH3180)
- Return NA when using `Series.str[...]` for values that are not long enough (GH3223)
- Display cursor coordinate information in time-series plots (GH1670)
- `to_html()` now accepts an optional “escape” argument to control reserved HTML character escaping (enabled by default) and escapes `&`, in addition to `<` and `>`. (GH2919)

### 38.26.3 API Changes

- Do not automatically upcast numeric specified dtypes to `int64` or `float64` (GH622 and GH797)

- DataFrame construction of lists and scalars, with no dtype present, will result in casting to `int64` or `float64`, regardless of platform. This is not an apparent change in the API, but noting it.
- Guarantee that `convert_objects()` for Series/DataFrame always returns a copy
- groupby operations will respect dtypes for numeric float operations (`float32/float64`); other types will be operated on, and will try to cast back to the input dtype (e.g. if an int is passed, as long as the output doesn't have nans, then an int will be returned)
- `backfill/pad/take/diff/ohlc` will now support `float32/int16/int8` operations
- Block types will upcast as needed in where/masking operations ([GH2793](#))
- Series now automatically will try to set the correct dtype based on passed datetimelike objects (date-time/Timestamp)
  - `timedelta64` are returned in appropriate cases (e.g. Series - Series, when both are `datetime64`)
  - mixed datetimes and objects ([GH2751](#)) in a constructor will be cast correctly
  - astype on datetimes to object are now handled (as well as NaT conversions to `np.nan`)
  - all timedelta like objects will be correctly assigned to `timedelta64` with mixed NaN and/or NaT allowed
- arguments to `DataFrame.clip` were inconsistent to NumPy and Series clipping ([GH2747](#))
- `util.testing.assert_frame_equal` now checks the column and index names ([GH2964](#))
- Constructors will now return a more informative `ValueError` on failures when invalid shapes are passed
- Don't suppress `TypeError` in `GroupBy.agg` ([GH3238](#))
- Methods return `None` when `inplace=True` ([GH1893](#))
- `HDFStore`
  - added the method `select_column` to select a single column from a table as a Series.
  - deprecated the `unique` method, can be replicated by `select_column(key, column).unique()`
  - `min_itemsize` parameter will now automatically create data\_columns for passed keys
- Downcast on pivot if possible ([GH3283](#)), adds argument `downcast` to `fillna`
- Introduced options `display.height/width` for explicitly specifying terminal height/width in characters. Deprecated `display.line_width`, now replaced by `display.width`. These defaults are in effect for scripts as well, so unless disabled, previously very wide output will now be output as "expand\_repr" style wrapped output.
- Various defaults for options (including `display.max_rows`) have been revised, after a brief survey concluded they were wrong for everyone. Now at `w=80,h=60`.
- HTML repr output in IPython qtconsole is once again controlled by the option `display.notebook_repr_html`, and on by default.

### 38.26.4 Bug Fixes

- Fix seg fault on empty data frame when `fillna` with `pad` or `backfill` ([GH2778](#))
- Single element ndarrays of datetimelike objects are handled (e.g. `np.array(datetime(2001,1,1,0,0))`), w/o dtype being passed
- 0-dim ndarrays with a passed dtype are handled correctly (e.g. `np.array(0.,dtype='float32')`)
- Fix some boolean indexing inconsistencies in `Series.__getitem__/_setitem__` ([GH2776](#))

- Fix issues with DataFrame and Series constructor with integers that overflow `int64` and some mixed typed type lists ([GH2845](#))
- `HDFStore`
  - Fix weird PyTables error when using too many selectors in a where also correctly filter on any number of values in a Term expression (so not using `numexpr` filtering, but `isin` filtering)
  - Internally, change all variables to be private-like (now have leading underscore)
  - Fixes for query parsing to correctly interpret boolean and `!=` ([GH2849](#), [GH2973](#))
  - Fixes for pathological case on `SparseSeries` with 0-len array and compression ([GH2931](#))
  - Fixes bug with writing rows if part of a block was all-nan ([GH3012](#))
  - Exceptions are now `ValueError` or `TypeError` as needed
  - A table will now raise if `min_itemsize` contains fields which are not queryables
- Bug showing up in `applymap` where some object type columns are converted ([GH2909](#)) had an incorrect default in `convert_objects`
- `TimeDeltas`
  - Series ops with a Timestamp on the rhs was throwing an exception ([GH2898](#)) added tests for Series ops with datetimes, timedeltas, Timestamps, and datelike Series on both lhs and rhs
  - Fixed subtle `timedelta64` inference issue on py3 & NumPy 1.7.0 ([GH3094](#))
  - Fixed some formatting issues on `timedelta` when negative
  - Support null checking on `timedelta64`, representing (and formatting) with `NaT`
  - Support `setitem` with `np.nan` value, converts to `NaT`
  - Support `min/max` ops in a DataFrame (`abs` not working, nor do we error on non-supported ops)
  - Support `idxmin/idxmax/abs/max/min` in a Series ([GH2989](#), [GH2982](#))
- Bug on in-place putmasking on an integer series that needs to be converted to float ([GH2746](#))
- Bug in `argsort` of `datetime64[ns]` Series with `NaT` ([GH2967](#))
- Bug in `value_counts` of `datetime64[ns]` Series ([GH3002](#))
- Fixed printing of `NaT` in an index
- Bug in `idxmin/idxmax` of `datetime64[ns]` Series with `NaT` ([GH2982](#))
- Bug in `icol`, `take` with negative indicies was producing incorrect return values (see [GH2922](#), [GH2892](#)), also check for out-of-bounds indices ([GH3029](#))
- Bug in DataFrame column insertion when the column creation fails, existing frame is left in an irrecoverable state ([GH3010](#))
- Bug in DataFrame update, `combine_first` where non-specified values could cause dtype changes ([GH3016](#), [GH3041](#))
- Bug in `groupby` with `first/last` where dtypes could change ([GH3041](#), [GH2763](#))
- Formatting of an index that has `nan` was inconsistent or wrong (would fill from other values), ([GH2850](#))
- Unstack of a frame with no nans would always cause dtype upcasting ([GH2929](#))
- Fix scalar `datetime.datetime` parsing bug in `read_csv` ([GH3071](#))
- Fixed slow printing of large Dataframes, due to inefficient dtype reporting ([GH2807](#))

- Fixed a segfault when using a function as grouper in groupby (GH3035)
- Fix pretty-printing of infinite data structures (closes GH2978)
- Fixed exception when plotting timeseries bearing a timezone (closes GH2877)
- str.contains ignored na argument (GH2806)
- Substitute warning for segfault when grouping with categorical grouper of mismatched length (GH3011)
- Fix exception in SparseSeries.density (GH2083)
- Fix upsampling bug with closed='left' and daily to daily data (GH3020)
- Fixed missing tick bars on scatter\_matrix plot (GH3063)
- Fixed bug in Timestamp(d,tz=foo) when d is date() rather than datetime() (GH2993)
- series.plot(kind='bar') now respects pylab color schem (GH3115)
- Fixed bug in reshape if not passed correct input, now raises TypeError (GH2719)
- Fixed a bug where Series ctor did not respect ordering if OrderedDict passed in (GH3282)
- Fix NameError issue on RESO\_US (GH2787)
- Allow selection in an *unordered* timeseries to work similarly to an *ordered* timeseries (GH2437).
- Fix implemented .xs when called with axes=1 and a level parameter (GH2903)
- Timestamp now supports the class method fromordinal similar to datetimes (GH3042)
- Fix issue with indexing a series with a boolean key and specifying a 1-len list on the rhs (GH2745) or a list on the rhs (GH3235)
- Fixed bug in groupby apply when kernel generate list of arrays having unequal len (GH1738)
- fixed handling of rolling\_corr with center=True which could produce corr>1 (GH3155)
- Fixed issues where indices can be passed as 'index/column' in addition to 0/1 for the axis parameter
- PeriodIndex.tolist now boxes to Period (GH3178)
- PeriodIndex.get\_loc KeyError now reports Period instead of ordinal (GH3179)
- df.to\_records bug when handling MultiIndex (GH3189)
- Fix Series.\_\_getitem\_\_ segfault when index less than -length (GH3168)
- Fix bug when using Timestamp as a date parser (GH2932)
- Fix bug creating date range from Timestamp with time zone and passing same time zone (GH2926)
- Add comparison operators to Period object (GH2781)
- Fix bug when concatenating two Series into a DataFrame when they have the same name (GH2797)
- Fix automatic color cycling when plotting consecutive timeseries without color arguments (GH2816)
- fixed bug in the pickling of PeriodIndex (GH2891)
- Upcast/split blocks when needed in a mixed DataFrame when setitem with an indexer (GH3216)
- Invoking df.applymap on a dataframe with dupe cols now raises a ValueError (GH2786)
- Apply with invalid returned indices raise correct Exception (GH2808)
- Fixed a bug in plotting log-scale bar plots (GH3247)
- df.plot() grid on/off now obeys the mpl default style, just like series.plot(). (GH3233)

- Fixed a bug in the legend of `plotting.andrews_curves()` (GH3278)
- Produce a series on apply if we only generate a singular series and have a simple index (GH2893)
- Fix Python ASCII file parsing when integer falls outside of floating point spacing (GH3258)
- fixed pretty printing of sets (GH3294)
- `Panel()` and `Panel.from_dict()` now respects ordering when give `OrderedDict` (GH3303)
- `DataFrame` where with a datetimelike incorrectly selecting (GH3311)
- Ensure index casts work even in `Int64Index`
- Fix `set_index` segfault when passing `MultiIndex` (GH3308)
- Ensure pickles created in py2 can be read in py3
- Insert ellipsis in `MultiIndex` summary repr (GH3348)
- `Groupby` will handle mutation among an input groups columns (and fallback to non-fast apply) (GH3380)
- Eliminated unicode errors on FreeBSD when using MPL GTK backend (GH3360)
- `Period.strftime` should return unicode strings always (GH3363)
- Respect passed `read_*` chunksize in `get_chunk` function (GH3406)

## 38.27 pandas 0.10.1

**Release date:** 2013-01-22

### 38.27.1 New Features

- Add data interface to World Bank WDI `pandas.io.wb` (GH2592)

### 38.27.2 API Changes

- Restored `inplace=True` behavior returning self (same object) with deprecation warning until 0.11 (GH1893)
- `HDFStore`
  - refactored `HDFStore` to deal with non-table stores as objects, will allow future enhancements
  - removed keyword `compression` from `put` (replaced by keyword `complib` to be consistent across library)
  - warn *PerformanceWarning* if you are attempting to store types that will be pickled by `PyTables`

### 38.27.3 Improvements to existing features

- `HDFStore`
  - enables storing of multi-index dataframes (closes GH1277)
  - support data column indexing and selection, via `data_columns` keyword in `append`
  - support write chunking to reduce memory footprint, via `chunksize` keyword to `append`
  - support automagic indexing via `index` keyword to `append`

- support `expectedrows` keyword in `append` to inform `PyTables` about the expected tablesize
- support `start` and `stop` keywords in `select` to limit the row selection space
- added `get_store` context manager to automatically import with `pandas`
- added column filtering via `columns` keyword in `select`
- added methods `append_to_multiple/select_as_multiple/select_as_coordinates` to do multiple-table `append`/`selection`
- added support for `datetime64` in `columns`
- added method `unique` to select the unique values in an indexable or data column
- added method `copy` to copy an existing store (and possibly upgrade)
- show the shape of the data on disk for non-table stores when printing the store
- added ability to read `PyTables` flavor tables (allows compatibility to other `HDF5` systems)
- Add `logx` option to `DataFrame/Series.plot` ([GH2327](#), [GH2565](#))
- Support reading gzipped data from file-like object
- `pivot_table` `aggfunc` can be anything used in `GroupBy.aggregate` ([GH2643](#))
- Implement `DataFrame` merges in case where set cardinalities might overflow 64-bit integer ([GH2690](#))
- Raise exception in C file parser if integer dtype specified and have NA values. ([GH2631](#))
- Attempt to parse ISO8601 format dates when `parse_dates=True` in `read_csv` for major performance boost in such cases ([GH2698](#))
- Add methods `neg` and `inv` to `Series`
- Implement `kind` option in `ExcelFile` to indicate whether it's an XLS or XLSX file ([GH2613](#))
- Documented a fast-path in `pd.read_csv` when parsing iso8601 datetime strings yielding as much as a 20x speedup. ([GH5993](#))

### 38.27.4 Bug Fixes

- Fix `read_csv/read_table` multithreading issues ([GH2608](#))
- `HDFStore`
  - correctly handle `nan` elements in string columns; serialize via the `nan_rep` keyword to `append`
  - raise correctly on non-implemented column types (unicode/date)
  - handle correctly `Term` passed types (e.g. `index<1000`, when `index` is `Int64`), (closes [GH512](#))
  - handle `Timestamp` correctly in `data_columns` (closes [GH2637](#))
  - contains correctly matches on non-natural names
  - correctly store `float32` dtypes in tables (if not other float types in the same table)
- Fix `DataFrame.info` bug with UTF8-encoded columns. ([GH2576](#))
- Fix `DatetimeIndex` handling of `FixedOffset` tz ([GH2604](#))
- More robust detection of being in IPython session for wide `DataFrame` console formatting ([GH2585](#))
- Fix platform issues with `file:///` in unit test ([GH2564](#))
- Fix bug and possible segfault when grouping by hierarchical level that contains NA values ([GH2616](#))

- Ensure that MultiIndex tuples can be constructed with NAs ([GH2616](#))
- Fix int64 overflow issue when unstacking MultiIndex with many levels ([GH2616](#))
- Exclude non-numeric data from DataFrame.quantile by default ([GH2625](#))
- Fix a Cython C int64 boxing issue causing read\_csv to return incorrect results ([GH2599](#))
- Fix groupby summing performance issue on boolean data ([GH2692](#))
- Don't bork Series containing datetime64 values with to\_datetime ([GH2699](#))
- Fix DataFrame.from\_records corner case when passed columns, index column, but empty record list ([GH2633](#))
- Fix C parser-tokenizer bug with trailing fields. ([GH2668](#))
- Don't exclude non-numeric data from GroupBy.max/min ([GH2700](#))
- Don't lose time zone when calling DatetimeIndex.drop ([GH2621](#))
- Fix setitem on a Series with a boolean key and a non-scalar as value ([GH2686](#))
- Box datetime64 values in Series.apply/map ([GH2627](#), [GH2689](#))
- Upconvert datetime + datetime64 values when concatenating frames ([GH2624](#))
- Raise a more helpful error message in merge operations when one DataFrame has duplicate columns ([GH2649](#))
- Fix partial date parsing issue occurring only when code is run at EOM ([GH2618](#))
- Prevent MemoryError when using counting sort in sortlevel with high-cardinality MultiIndex objects ([GH2684](#))
- Fix Period resampling bug when all values fall into a single bin ([GH2070](#))
- Fix buggy interaction with usecols argument in read\_csv when there is an implicit first index column ([GH2654](#))
- Fix bug in Index.summary() where string format methods were being called incorrectly. ([GH3869](#))

## 38.28 pandas 0.10.0

**Release date:** 2012-12-17

### 38.28.1 New Features

- Brand new high-performance delimited file parsing engine written in C and Cython. 50% or better performance in many standard use cases with a fraction as much memory usage. ([GH407](#), [GH821](#))
- Many new file parser (read\_csv, read\_table) features:
  - Support for on-the-fly gzip or bz2 decompression (*compression* option)
  - Ability to get back numpy.recarray instead of DataFrame (*as\_reccarray=True*)
  - *dtype* option: explicit column dtypes
  - *usecols* option: specify list of columns to be read from a file. Good for reading very wide files with many irrelevant columns ([GH1216](#) [GH926](#), [GH2465](#))
  - Enhanced unicode decoding support via *encoding* option
  - *skipinitialspace* dialect option
  - Can specify strings to be recognized as True (*true\_values*) or False (*false\_values*)



- High-performance *delim\_whitespace* option for whitespace-delimited files; a preferred alternative to the 's+' regular expression delimiter
- Option to skip “bad” lines (wrong number of fields) that would otherwise have caused an error in the past (*error\_bad\_lines* and *warn\_bad\_lines* options)
- Substantially improved performance in the parsing of integers with thousands markers and lines with comments
- Easy of European (and other) decimal formats (*decimal* option) (GH584, GH2466)
- Custom line terminators (e.g. *lineterminator*='~') (GH2457)
- Handling of no trailing commas in CSV files (GH2333)
- Ability to handle fractional seconds in *date\_converters* (GH2209)
- *read\_csv* allow scalar arg to *na\_values* (GH1944)
- Explicit column dtype specification in *read\_\** functions (GH1858)
- Easier CSV dialect specification (GH1743)
- Improve parser performance when handling special characters (GH1204)
- Google Analytics API integration with easy oauth2 workflow (GH2283)
- Add error handling to *Series.str.encode/decode* (GH2276)
- Add *where* and *mask* to *Series* (GH2337)
- Grouped histogram via *by* keyword in *Series/DataFrame.hist* (GH2186)
- Support optional *min\_periods* keyword in *corr* and *cov* for both *Series* and *DataFrame* (GH2002)
- Add *duplicated* and *drop\_duplicates* functions to *Series* (GH1923)
- Add docs for *HDFStore table* format
- ‘density’ property in *SparseSeries* (GH2384)
- Add *ffill* and *bfill* convenience functions for forward- and backfilling time series data (GH2284)
- New option configuration system and functions *set\_option*, *get\_option*, *describe\_option*, and *reset\_option*. Deprecate *set\_printoptions* and *reset\_printoptions* (GH2393). You can also access options as attributes via *pandas.options.X*
- Wide *DataFrames* can be viewed more easily in the console with new *expand\_frame\_repr* and *line\_width* configuration options. This is on by default now (GH2436)
- Scikits.timeseries-like moving window functions via *rolling\_window* (GH1270)

### 38.28.2 Experimental Features

- Add support for *Panel4D*, a named 4 Dimensional structure
- Add support for *ndpanel* factory functions, to create custom, domain-specific N-Dimensional containers

### 38.28.3 API Changes

- The default binning/labeling behavior for *resample* has been changed to *closed*='left', *label*='left' for daily and lower frequencies. This had been a large source of confusion for users. See “what’s new” page for more on this. (GH2410)

- Methods with `inplace` option now return `None` instead of the calling (modified) object ([GH1893](#))
- The special case `DataFrame - TimeSeries` doing column-by-column broadcasting has been deprecated. Users should explicitly do e.g. `df.sub(ts, axis=0)` instead. This is a legacy hack and can lead to subtle bugs.
- `inf/-inf` are no longer considered as NA by `isnull/notnull`. To be clear, this is legacy cruft from early pandas. This behavior can be globally re-enabled using the new option `mode.use_inf_as_null` ([GH2050](#), [GH1919](#))
- `pandas.merge` will now default to `sort=False`. For many use cases sorting the join keys is not necessary, and doing it by default is wasteful
- Specify `header=0` explicitly to replace existing column names in file in `read_*` functions.
- Default column names for header-less parsed files (yielded by `read_csv`, etc.) are now the integers `0, 1, ...`. A new argument `prefix` has been added; to get the v0.9.x behavior specify `prefix='X'` ([GH2034](#)). This API change was made to make the default column names more consistent with the `DataFrame` constructor's default column names when none are specified.
- `DataFrame` selection using a boolean frame now preserves input shape
- If function passed to `Series.apply` yields a `Series`, result will be a `DataFrame` ([GH2316](#))
- Values like YES/NO/yes/no will not be considered as boolean by default any longer in the file parsers. This can be customized using the new `true_values` and `false_values` options ([GH2360](#))
- `obj.fillna()` is no longer valid; make `method='pad'` no longer the default option, to be more explicit about what kind of filling to perform. Add `ffill/bfill` convenience functions per above ([GH2284](#))
- `HDFStore.keys()` now returns an absolute path-name for each key
- `to_string()` now always returns a unicode string. ([GH2224](#))
- File parsers will not handle NA sentinel values arising from passed converter functions

### 38.28.4 Improvements to existing features

- Add `nrows` option to `DataFrame.from_records` for iterators ([GH1794](#))
- Unstack/reshape algorithm rewrite to avoid high memory use in cases where the number of observed key-tuples is much smaller than the total possible number that could occur ([GH2278](#)). Also improves performance in most cases.
- Support duplicate columns in `DataFrame.from_records` ([GH2179](#))
- Add `normalize` option to `Series/DataFrame.asfreq` ([GH2137](#))
- `SparseSeries` and `SparseDataFrame` construction from empty and scalar values now no longer create dense `ndarrays` unnecessarily ([GH2322](#))
- `HDFStore` now supports hierarchical keys ([GH2397](#))
- Support multiple query selection formats for `HDFStore` tables ([GH1996](#))
- Support `del store['df']` syntax to delete `HDFStores`
- Add multi-dtype support for `HDFStore` tables
- `min_itemsize` parameter can be specified in `HDFStore` table creation
- Indexing support in `HDFStore` tables ([GH698](#))
- Add `line_terminator` option to `DataFrame.to_csv` ([GH2383](#))
- added implementation of `str(x)/unicode(x)/bytes(x)` to major pandas data structures, which should do the right thing on both `py2.x` and `py3.x`. ([GH2224](#))

- Reduce `groupby.apply` overhead substantially by low-level manipulation of internal NumPy arrays in DataFrames (GH535)
- Implement `value_vars` in `melt` and add `melt` to pandas namespace (GH2412)
- Added boolean comparison operators to Panel
- Enable `Series.str.strip/lstrip/rstrip` methods to take an argument (GH2411)
- The DataFrame ctor now respects column ordering when given an `OrderedDict` (GH2455)
- Assigning `DatetimeIndex` to `Series` changes the class to `TimeSeries` (GH2139)
- Improve performance of `.value_counts` method on non-integer data (GH2480)
- `get_level_values` method for `MultiIndex` return `Index` instead of `ndarray` (GH2449)
- `convert_to_r_dataframe` conversion for datetime values (GH2351)
- Allow `DataFrame.to_csv` to represent `inf` and `nan` differently (GH2026)
- Add `min_i` argument to `nancorr` to specify minimum required observations (GH2002)
- Add `inplace` option to `sortlevel / sort` functions on DataFrame (GH1873)
- Enable DataFrame to accept scalar constructor values like `Series` (GH1856)
- `DataFrame.from_records` now takes optional `size` parameter (GH1794)
- include iris dataset (GH1709)
- No `datetime64` DataFrame column conversion of `datetime.datetime` with `tzinfo` (GH1581)
- Micro-optimizations in DataFrame for tracking state of internal consolidation (GH217)
- Format parameter in `DataFrame.to_csv` (GH1525)
- Partial string slicing for `DatetimeIndex` for daily and higher frequencies (GH2306)
- Implement `col_space` parameter in `to_html` and `to_string` in DataFrame (GH1000)
- Override `Series.tolist` and box `datetime64` types (GH2447)
- Optimize `unstack` memory usage by compressing indices (GH2278)
- Fix HTML repr in IPython qtconsole if opening window is small (GH2275)
- Escape more special characters in console output (GH2492)
- `df.select` now invokes `bool` on the result of `crit(x)` (GH2487)

### 38.28.5 Bug Fixes

- Fix major performance regression in `DataFrame.iteritems` (GH2273)
- Fixes bug when negative period passed to `Series/DataFrame.diff` (GH2266)
- Escape tabs in console output to avoid alignment issues (GH2038)
- Properly box `datetime64` values when retrieving cross-section from mixed-dtype DataFrame (GH2272)
- Fix concatenation bug leading to GH2057, GH2257
- Fix regression in `Index` console formatting (GH2319)
- Box `Period` data when assigning `PeriodIndex` to frame column (GH2243, GH2281)
- Raise exception on calling `reset_index` on `Series` with `inplace=True` (GH2277)

- Enable setting multiple columns in DataFrame with hierarchical columns (GH2295)
- Respect dtype=object in DataFrame constructor (GH2291)
- Fix DatetimeIndex.join bug with tz-aware indexes and how='outer' (GH2317)
- pop(...) and del works with DataFrame with duplicate columns (GH2349)
- Treat empty strings as NA in date parsing (rather than let dateutil do something weird) (GH2263)
- Prevent uint64 -> int64 overflows (GH2355)
- Enable joins between MultiIndex and regular Index (GH2024)
- Fix time zone metadata issue when unioning non-overlapping DatetimeIndex objects (GH2367)
- Raise/handle int64 overflows in parsers (GH2247)
- Deleting of consecutive rows in HDFStore tables` is much faster than before
- Appending on a HDFStore would fail if the table was not first created via put
- Use col\_space argument as minimum column width in DataFrame.to\_html (GH2328)
- Fix tz-aware DatetimeIndex.to\_period (GH2232)
- Fix DataFrame row indexing case with MultiIndex (GH2314)
- Fix to\_excel exporting issues with Timestamp objects in index (GH2294)
- Fixes assigning scalars and array to hierarchical column chunk (GH1803)
- Fixed a UnicodeDecodeError with series tidy\_repr (GH2225)
- Fixed issued with duplicate keys in an index (GH2347, GH2380)
- Fixed issues re: Hash randomization, default on starting w/ py3.3 (GH2331)
- Fixed issue with missing attributes after loading a pickled dataframe (GH2431)
- Fix Timestamp formatting with tzoffset time zone in dateutil 2.1 (GH2443)
- Fix GroupBy.apply issue when using BinGrouper to do ts binning (GH2300)
- Fix issues resulting from datetime.datetime columns being converted to datetime64 when calling DataFrame.apply. (GH2374)
- Raise exception when calling to\_panel on non uniquely-indexed frame (GH2441)
- Improved detection of console encoding on IPython zmq frontends (GH2458)
- Preserve time zone when .append-ing two time series (GH2260)
- Box timestamps when calling reset\_index on time-zone-aware index rather than creating a tz-less datetime64 column (GH2262)
- Enable searching non-string columns in DataFrame.filter(like=...) (GH2467)
- Fixed issue with losing nanosecond precision upon conversion to DatetimeIndex (GH2252)
- Handle timezones in Datetime.normalize (GH2338)
- Fix test case where dtype specification with endianness causes failures on big endian machines (GH2318)
- Fix plotting bug where upsampling causes data to appear shifted in time (GH2448)
- Fix read\_csv failure for UTF-16 with BOM and skiprows (GH2298)
- read\_csv with names arg not implicitly setting header=None (GH2459)
- Unrecognized compression mode causes segfault in read\_csv (GH2474)

- In `read_csv`, `header=0` and passed names should discard first row (GH2269)
- Correctly route to `stdout/stderr` in `read_table` (GH2071)
- Fix exception when `Timestamp.to_datetime` is called on a `Timestamp` with `tzoffset` (GH2471)
- Fixed unintentional conversion of `datetime64` to `long` in `groupby.first()` (GH2133)
- Union of empty `DataFrames` now return empty with concatenated index (GH2307)
- `DataFrame.sort_index` raises more helpful exception if sorting by column with duplicates (GH2488)
- `DataFrame.to_string` formatters can be list, too (GH2520)
- `DataFrame.combine_first` will always result in the union of the index and columns, even if one `DataFrame` is length-zero (GH2525)
- Fix several `DataFrame.icol/irow` with duplicate indices issues (GH2228, GH2259)
- Use `Series` names for column names when using `concat` with `axis=1` (GH2489)
- Raise `Exception` if `start`, `end`, `periods` all passed to `date_range` (GH2538)
- Fix `Panel` resampling issue (GH2537)

## 38.29 pandas 0.9.1

**Release date:** 2012-11-14

### 38.29.1 New Features

- Can specify multiple sort orders in `DataFrame/Series.sort/sort_index` (GH928)
- New *top* and *bottom* options for handling NAs in `rank` (GH1508, GH2159)
- Add *where* and *mask* functions to `DataFrame` (GH2109, GH2151)
- Add *at\_time* and *between\_time* functions to `DataFrame` (GH2149)
- Add flexible *pow* and *rpow* methods to `DataFrame` (GH2190)

### 38.29.2 API Changes

- Upsampling period index “spans” intervals. Example: annual periods upsampled to monthly will span all months in each year
- `Period.end_time` will yield timestamp at last nanosecond in the interval (GH2124, GH2125, GH1764)
- File parsers no longer coerce to float or bool for columns that have custom converters specified (GH2184)

### 38.29.3 Improvements to existing features

- Time rule inference for week-of-month (e.g. WOM-2FRI) rules (GH2140)
- Improve performance of `datetime` + business day offset with large number of offset periods
- Improve HTML display of `DataFrame` objects with hierarchical columns
- Enable referencing of Excel columns by their column names (GH1936)

- `DataFrame.dot` can accept `ndarrays` (GH2042)
- Support negative periods in `Panel.shift` (GH2164)
- Make `.drop(...)` work with non-unique indexes (GH2101)
- Improve performance of `Series/DataFrame.diff` (re: GH2087)
- Support unary `~` (`__invert__`) in `DataFrame` (GH2110)
- Turn off pandas-style tick locators and formatters (GH2205)
- `DataFrame[DataFrame]` uses `DataFrame.where` to compute masked frame (GH2230)

### 38.29.4 Bug Fixes

- Fix some duplicate-column `DataFrame` constructor issues (GH2079)
- Fix bar plot color cycle issues (GH2082)
- Fix off-center grid for stacked bar plots (GH2157)
- Fix plotting bug if inferred frequency is offset with  $N > 1$  (GH2126)
- Implement comparisons on date offsets with fixed delta (GH2078)
- Handle `inf/-inf` correctly in `read_*` parser functions (GH2041)
- Fix matplotlib unicode interaction bug
- Make WLS r-squared match statsmodels 0.5.0 fixed value
- Fix zero-trimming `DataFrame` formatting bug
- Correctly compute/box `datetime64` min/max values from `Series.min/max` (GH2083)
- Fix unstacking edge case with unrepresented groups (GH2100)
- Fix `Series.str` failures when using pipe pattern `'|'` (GH2119)
- Fix pretty-printing of dict entries in `Series`, `DataFrame` (GH2144)
- Cast other `datetime64` values to nanoseconds in `DataFrame` ctor (GH2095)
- Alias `Timestamp.astimezone` to `tz_convert`, so will yield `Timestamp` (GH2060)
- Fix `timedelta64` formatting from `Series` (GH2165, GH2146)
- Handle `None` values gracefully in dict passed to `Panel` constructor (GH2075)
- Box `datetime64` values as `Timestamp` objects in `Series/DataFrame.i`get (GH2148)
- Fix `Timestamp` indexing bug in `DatetimeIndex.insert` (GH2155)
- Use index name(s) (if any) in `DataFrame.to_records` (GH2161)
- Don't lose index names in `Panel.to_frame/DataFrame.to_panel` (GH2163)
- Work around length-0 boolean indexing NumPy bug (GH2096)
- Fix partial integer indexing bug in `DataFrame.xs` (GH2107)
- Fix variety of `cut/qcut` string-bin formatting bugs (GH1978, GH1979)
- Raise `Exception` when `xs` view not possible of `MultiIndex'd DataFrame` (GH2117)
- Fix `groupby(...).first()` issue with `datetime64` (GH2133)
- Better floating point error robustness in some `rolling_*` functions (GH2114, GH2527)

- Fix ewma NA handling in the middle of Series (GH2128)
- Fix numerical precision issues in diff with integer data (GH2087)
- Fix bug in MultiIndex.\_\_getitem\_\_ with NA values (GH2008)
- Fix DataFrame.from\_records dict-arg bug when passing columns (GH2179)
- Fix Series and DataFrame.diff for integer dtypes (GH2087, GH2174)
- Fix bug when taking intersection of DatetimeIndex with empty index (GH2129)
- Pass through timezone information when calling DataFrame.align (GH2127)
- Properly sort when joining on datetime64 values (GH2196)
- Fix indexing bug in which False/True were being coerced to 0/1 (GH2199)
- Many unicode formatting fixes (GH2201)
- Fix improper MultiIndex conversion issue when assigning e.g. DataFrame.index (GH2200)
- Fix conversion of mixed-type DataFrame to ndarray with dup columns (GH2236)
- Fix duplicate columns issue (GH2218, GH2219)
- Fix SparseSeries.\_\_pow\_\_ issue with NA input (GH2220)
- Fix icol with integer sequence failure (GH2228)
- Fixed resampling tz-aware time series issue (GH2245)
- SparseDataFrame.icol was not returning SparseSeries (GH2227, GH2229)
- Enable ExcelWriter to handle PeriodIndex (GH2240)
- Fix issue constructing DataFrame from empty Series with name (GH2234)
- Use console-width detection in interactive sessions only (GH1610)
- Fix parallel\_coordinates legend bug with mpl 1.2.0 (GH2237)
- Make tz\_localize work in corner case of empty Series (GH2248)

## 38.30 pandas 0.9.0

**Release date:** 10/7/2012

### 38.30.1 New Features

- Add `str.encode` and `str.decode` to Series (GH1706)
- Add `to_latex` method to DataFrame (GH1735)
- Add convenient expanding window equivalents of all rolling\_\* ops (GH1785)
- Add Options class to pandas.io.data for fetching options data from Yahoo! Finance (GH1748, GH1739)
- Recognize and convert more boolean values in file parsing (Yes, No, TRUE, FALSE, variants thereof) (GH1691, GH1295)
- Add Panel.update method, analogous to DataFrame.update (GH1999, GH1988)



### 38.30.2 Improvements to existing features

- Proper handling of NA values in merge operations ([GH1990](#))
- Add `flags` option for `re.compile` in some `Series.str` methods ([GH1659](#))
- Parsing of UTC date strings in `read_*` functions ([GH1693](#))
- Handle generator input to `Series` ([GH1679](#))
- Add `na_action='ignore'` to `Series.map` to quietly propagate NAs ([GH1661](#))
- Add `args/kwds` options to `Series.apply` ([GH1829](#))
- Add `inplace` option to `Series/DataFrame.reset_index` ([GH1797](#))
- Add `level` parameter to `Series.reset_index`
- Add quoting option for `DataFrame.to_csv` ([GH1902](#))
- Indicate long column value truncation in `DataFrame` output with `...` ([GH1854](#))
- `DataFrame.dot` will not do data alignment, and also work with `Series` ([GH1915](#))
- Add `na` option for missing data handling in some vectorized string methods ([GH1689](#))
- If `index_label=False` in `DataFrame.to_csv`, do not print fields/commas in the text output. Results in easier importing into R ([GH1583](#))
- Can pass tuple/list of axes to `DataFrame.dropna` to simplify repeated calls (dropping both columns and rows) ([GH924](#))
- Improve `DataFrame.to_html` output for hierarchically-indexed rows (do not repeat levels) ([GH1929](#))
- `TimeSeries.between_time` can now select times across midnight ([GH1871](#))
- Enable `skip_footer` parameter in `ExcelFile.parse` ([GH1843](#))

### 38.30.3 API Changes

- Change default header names in `read_*` functions to more Pythonic `X0`, `X1`, etc. instead of `X.1`, `X.2`. ([GH2000](#))
- Deprecated `day_of_year` API removed from `PeriodIndex`, use `dayofyear` ([GH1723](#))
- Don't modify NumPy suppress printoption at import time
- The internal HDF5 data arrangement for `DataFrames` has been transposed. Legacy files will still be readable by `HDFStore` ([GH1834](#), [GH1824](#))
- Legacy cruft removed: `pandas.stats.misc.quantileTS`
- Use ISO8601 format for `Period` repr: monthly, daily, and on down ([GH1776](#))
- Empty `DataFrame` columns are now created as object dtype. This will prevent a class of `TypeError`s that was occurring in code where the dtype of a column would depend on the presence of data or not (e.g. a SQL query having results) ([GH1783](#))
- Setting parts of `DataFrame/Panel` using `ix` now aligns input `Series/DataFrame` ([GH1630](#))
- `first` and `last` methods in `GroupBy` no longer drop non-numeric columns ([GH1809](#))
- Resolved inconsistencies in specifying custom NA values in text parser. `na_values` of type dict no longer override default NAs unless `keep_default_na` is set to false explicitly ([GH1657](#))
- Enable `skipfooter` parameter in text parsers as an alias for `skip_footer`



### 38.30.4 Bug Fixes

- Perform arithmetic column-by-column in mixed-type DataFrame to avoid type upcasting issues. Caused downstream DataFrame.diff bug (GH1896)
- Fix matplotlib auto-color assignment when no custom spectrum passed. Also respect passed color keyword argument (GH1711)
- Fix resampling logical error with closed='left' (GH1726)
- Fix critical DatetimeIndex.union bugs (GH1730, GH1719, GH1745, GH1702, GH1753)
- Fix critical DatetimeIndex.intersection bug with unanchored offsets (GH1708)
- Fix MM-YYYY time series indexing case (GH1672)
- Fix case where Categorical group key was not being passed into index in GroupBy result (GH1701)
- Handle Ellipsis in Series.\_\_getitem\_\_/\_setitem\_\_ (GH1721)
- Fix some bugs with handling datetime64 scalars of other units in NumPy 1.6 and 1.7 (GH1717)
- Fix performance issue in MultiIndex.format (GH1746)
- Fixed GroupBy bugs interacting with DatetimeIndex asof / map methods (GH1677)
- Handle factors with NAs in pandas.rpy (GH1615)
- Fix statsmodels import in pandas.stats.var (GH1734)
- Fix DataFrame repr/info summary with non-unique columns (GH1700)
- Fix Series.iget\_value for non-unique indexes (GH1694)
- Don't lose tzinfo when passing DatetimeIndex as DataFrame column (GH1682)
- Fix tz conversion with time zones that haven't had any DST transitions since first date in the array (GH1673)
- Fix field access with UTC->local conversion on unsorted arrays (GH1756)
- Fix isnull handling of array-like (list) inputs (GH1755)
- Fix regression in handling of Series in Series constructor (GH1671)
- Fix comparison of Int64Index with DatetimeIndex (GH1681)
- Fix min\_periods handling in new rolling\_max/min at array start (GH1695)
- Fix errors with how='median' and generic NumPy resampling in some cases caused by SeriesBinGrouper (GH1648, GH1688)
- When grouping by level, exclude unobserved levels (GH1697)
- Don't lose tzinfo in DatetimeIndex when shifting by different offset (GH1683)
- Hack to support storing data with a zero-length axis in HDFStore (GH1707)
- Fix DatetimeIndex tz-aware range generation issue (GH1674)
- Fix method='time' interpolation with intraday data (GH1698)
- Don't plot all-NA DataFrame columns as zeros (GH1696)
- Fix bug in scatter\_plot with by option (GH1716)
- Fix performance problem in infer\_freq with lots of non-unique stamps (GH1686)
- Fix handling of PeriodIndex as argument to create MultiIndex (GH1705)
- Fix re: unicode MultiIndex level names in Series/DataFrame repr (GH1736)

- Handle PeriodIndex in to\_datetime instance method ([GH1703](#))
- Support StaticTzInfo in DatetimeIndex infrastructure ([GH1692](#))
- Allow MultiIndex setops with length-0 other type indexes ([GH1727](#))
- Fix handling of DatetimeIndex in DataFrame.to\_records ([GH1720](#))
- Fix handling of general objects in isnull on which bool(...) fails ([GH1749](#))
- Fix .ix indexing with MultiIndex ambiguity ([GH1678](#))
- Fix .ix setting logic error with non-unique MultiIndex ([GH1750](#))
- Basic indexing now works on MultiIndex with > 1000000 elements, regression from earlier version of pandas ([GH1757](#))
- Handle non-float64 dtypes in fast DataFrame.corr/cov code paths ([GH1761](#))
- Fix DatetimeIndex.isin to function properly ([GH1763](#))
- Fix conversion of array of tz-aware datetime.datetime to DatetimeIndex with right time zone ([GH1777](#))
- Fix DST issues with generating anchored date ranges ([GH1778](#))
- Fix issue calling sort on result of Series.unique ([GH1807](#))
- Fix numerical issue leading to square root of negative number in rolling\_std ([GH1840](#))
- Let Series.str.split accept no arguments (like str.split) ([GH1859](#))
- Allow user to have dateutil 2.1 installed on a Python 2 system ([GH1851](#))
- Catch ImportError less aggressively in pandas/\_\_init\_\_.py ([GH1845](#))
- Fix pip source installation bug when installing from GitHub ([GH1805](#))
- Fix error when window size > array size in rolling\_apply ([GH1850](#))
- Fix pip source installation issues via SSH from GitHub
- Fix OLS.summary when column is a tuple ([GH1837](#))
- Fix bug in \_\_doc\_\_ patching when -OO passed to interpreter ([GH1792](#) [GH1741](#) [GH1774](#))
- Fix unicode console encoding issue in IPython notebook ([GH1782](#), [GH1768](#))
- Fix unicode formatting issue with Series.name ([GH1782](#))
- Fix bug in DataFrame.duplicated with datetime64 columns ([GH1833](#))
- Fix bug in Panel internals resulting in error when doing fillna after truncate not changing size of panel ([GH1823](#))
- Prevent segfault due to MultiIndex not being supported in HDFStore table format ([GH1848](#))
- Fix UnboundLocalError in Panel.\_\_setitem\_\_ and add better error ([GH1826](#))
- Fix to\_csv issues with list of string entries. Isnull works on list of strings now too ([GH1791](#))
- Fix Timestamp comparisons with datetime values outside the nanosecond range (1677-2262)
- Revert to prior behavior of normalize\_date with datetime.date objects (return datetime)
- Fix broken interaction between np.nansum and Series.any/all
- Fix bug with multiple column date parsers ([GH1866](#))
- DatetimeIndex.union(Int64Index) was broken
- Make plot x vs y interface consistent with integer indexing ([GH1842](#))

- `set_index` inplace modified data even if unique check fails (GH1831)
- Only use Q-OCT/NOV/DEC in quarterly frequency inference (GH1789)
- Upcast to `dtype=object` when unstacking boolean DataFrame (GH1820)
- Fix float64/float32 merging bug (GH1849)
- Fixes to `Period.start_time` for non-daily frequencies (GH1857)
- Fix failure when converter used on `index_col` in `read_csv` (GH1835)
- Implement `PeriodIndex.append` so that `pandas.concat` works correctly (GH1815)
- Avoid Cython out-of-bounds access causing segfault sometimes in `pad_2d`, `backfill_2d`
- Fix resampling error with intraday times and anchored target time (like AS-DEC) (GH1772)
- Fix `.ix` indexing bugs with mixed-integer indexes (GH1799)
- Respect passed `color` keyword argument in `Series.plot` (GH1890)
- Fix `rolling_min/max` when the window is larger than the size of the input array. Check other malformed inputs (GH1899, GH1897)
- Rolling variance / standard deviation with only a single observation in window (GH1884)
- Fix unicode sheet name failure in `to_excel` (GH1828)
- Override `DatetimeIndex.min/max` to return Timestamp objects (GH1895)
- Fix column name formatting issue in length-truncated column (GH1906)
- Fix broken handling of copying Index metadata to new instances created by `view(...)` calls inside the NumPy infrastructure
- Support `datetime.date` again in `DateOffset.rollback/rollforward`
- Raise Exception if `set` passed to `Series` constructor (GH1913)
- Add `TypeError` when appending `HDFStore` table w/ wrong index type (GH1881)
- Don't raise exception on empty inputs in EW functions (e.g. `ewma`) (GH1900)
- Make `asof` work correctly with `PeriodIndex` (GH1883)
- Fix extlinks in doc build
- Fill boolean DataFrame with NaN when calling `shift` (GH1814)
- Fix `setuptools` bug causing `pip` not to Cythonize `.pyx` files sometimes
- Fix negative integer indexing regression in `.ix` from 0.7.x (GH1888)
- Fix error while retrieving timezone and utc offset from subclasses of `datetime.tzinfo` without `.zone` and `._utcoffset` attributes (GH1922)
- Fix DataFrame formatting of small, non-zero FP numbers (GH1911)
- Various fixes by upcasting of `date` -> `datetime` (GH1395)
- Raise better exception when passing multiple functions with the same name, such as lambdas, to `GroupBy.aggregate`
- Fix DataFrame `apply` with `axis=1` on a non-unique index (GH1878)
- Proper handling of Index subclasses in `pandas.unique` (GH1759)
- Set index names in `DataFrame.from_records` (GH1744)

- Fix time series indexing error with duplicates, under and over hash table size cutoff ([GH1821](#))
- Handle list keys in addition to tuples in `DataFrame.xs` when partial-indexing a hierarchically-indexed `DataFrame` ([GH1796](#))
- Support multiple column selection in `DataFrame.__getitem__` with duplicate columns ([GH1943](#))
- Fix time zone localization bug causing improper fields (e.g. hours) in time zones that have not had a UTC transition in a long time ([GH1946](#))
- Fix errors when parsing and working with fixed offset timezones ([GH1922](#), [GH1928](#))
- Fix text parser bug when handling UTC datetime objects generated by `dateutil` ([GH1693](#))
- Fix plotting bug when 'B' is the inferred frequency but index actually contains weekends ([GH1668](#), [GH1669](#))
- Fix plot styling bugs ([GH1666](#), [GH1665](#), [GH1658](#))
- Fix plotting bug with index/columns with unicode ([GH1685](#))
- Fix `DataFrame` constructor bug when passed `Series` with `datetime64` dtype in a dict ([GH1680](#))
- Fixed regression in generating `DatetimeIndex` using timezone aware `datetime.datetime` ([GH1676](#))
- Fix `DataFrame` bug when printing concatenated `DataFrames` with duplicated columns ([GH1675](#))
- Fixed bug when plotting time series with multiple intraday frequencies ([GH1732](#))
- Fix bug in `DataFrame.duplicated` to enable iterables other than list-types as input argument ([GH1773](#))
- Fix resample bug when passed list of lambdas as *how* argument ([GH1808](#))
- Repr fix for `MultiIndex` level with all NAs ([GH1971](#))
- Fix `PeriodIndex` slicing bug when slice start/end are out-of-bounds ([GH1977](#))
- Fix `read_table` bug when parsing unicode ([GH1975](#))
- Fix `BlockManager.iget` bug when dealing with non-unique `MultiIndex` as columns ([GH1970](#))
- Fix `reset_index` bug if both `drop` and `level` are specified ([GH1957](#))
- Work around unsafe NumPy object->int casting with Cython function ([GH1987](#))
- Fix `datetime64` formatting bug in `DataFrame.to_csv` ([GH1993](#))
- Default start date in `pandas.io.data` to 1/1/2000 as the docs say ([GH2011](#))

## 38.31 pandas 0.8.1

**Release date:** July 22, 2012

### 38.31.1 New Features

- Add vectorized, NA-friendly string methods to `Series` ([GH1621](#), [GH620](#))
- Can pass dict of per-column line styles to `DataFrame.plot` ([GH1559](#))
- Selective plotting to secondary y-axis on same subplot ([GH1640](#))
- Add new `bootstrap_plot` plot function
- Add new `parallel_coordinates` plot function ([GH1488](#))
- Add `radviz` plot function ([GH1566](#))

- Add `multi_sparse` option to `set_printoptions` to modify display of hierarchical indexes (GH1538)
- Add `dropna` method to Panel (GH171)

### 38.31.2 Improvements to existing features

- Use moving min/max algorithms from Bottleneck in `rolling_min/rolling_max` for > 100x speedup. (GH1504, GH50)
- Add Cython group median method for >15x speedup (GH1358)
- Drastically improve `to_datetime` performance on ISO8601 datetime strings (with no time zones) (GH1571)
- Improve single-key groupby performance on large data sets, accelerate use of groupby with a Categorical variable
- Add ability to append hierarchical index levels with `set_index` and to drop single levels with `reset_index` (GH1569, GH1577)
- Always apply passed functions in `resample`, even if upsampling (GH1596)
- Avoid unnecessary copies in DataFrame constructor with explicit dtype (GH1572)
- Cleaner DatetimeIndex string representation with 1 or 2 elements (GH1611)
- Improve performance of array-of-Period to PeriodIndex, convert such arrays to PeriodIndex inside Index (GH1215)
- More informative string representation for weekly Period objects (GH1503)
- Accelerate 3-axis multi data selection from homogeneous Panel (GH979)
- Add `adjust` option to `ewma` to disable adjustment factor (GH1584)
- Add new matplotlib converters for high frequency time series plotting (GH1599)
- Handling of tz-aware datetime.datetime objects in `to_datetime`; raise Exception unless `utc=True` given (GH1581)

### 38.31.3 Bug Fixes

- Fix NA handling in `DataFrame.to_panel` (GH1582)
- Handle TypeError issues inside `PyObject_RichCompareBool` calls in `khash` (GH1318)
- Fix resampling bug to lower case daily frequency (GH1588)
- Fix kendall/spearman `DataFrame.corr` bug with no overlap (GH1595)
- Fix bug in `DataFrame.set_index` (GH1592)
- Don't ignore axes in boxplot if by specified (GH1565)
- Fix Panel `.ix` indexing with integers bug (GH1603)
- Fix Partial indexing bugs (years, months, ...) with PeriodIndex (GH1601)
- Fix MultiIndex console formatting issue (GH1606)
- Unordered index with duplicates doesn't yield scalar location for single entry (GH1586)
- Fix resampling of tz-aware time series with "anchored" freq (GH1591)
- Fix `DataFrame.rank` error on integer data (GH1589)
- Selection of multiple SparseDataFrame columns by list in `__getitem__` (GH1585)

- Override Index.tolist for compatibility with MultiIndex (GH1576)
- Fix hierarchical summing bug with MultiIndex of length 1 (GH1568)
- Work around numpy.concatenate use/bug in Series.set\_value (GH1561)
- Ensure Series/DataFrame are sorted before resampling (GH1580)
- Fix unhandled IndexError when indexing very large time series (GH1562)
- Fix DatetimeIndex intersection logic error with irregular indexes (GH1551)
- Fix unit test errors on Python 3 (GH1550)
- Fix .ix indexing bugs in duplicate DataFrame index (GH1201)
- Better handle errors with non-existing objects in HDFStore (GH1254)
- Don't copy int64 array data in DatetimeIndex when copy=False (GH1624)
- Fix resampling of conforming periods quarterly to annual (GH1622)
- Don't lose index name on resampling (GH1631)
- Support python-dateutil version 2.1 (GH1637)
- Fix broken scatter\_matrix axis labeling, esp. with time series (GH1625)
- Fix cases where extra keywords weren't being passed on to matplotlib from Series.plot (GH1636)
- Fix BusinessMonthBegin logic for dates before 1st bday of month (GH1645)
- Ensure string alias converted (valid in DatetimeIndex.get\_loc) in DataFrame.xs / \_\_getitem\_\_ (GH1644)
- Fix use of string alias timestamps with tz-aware time series (GH1647)
- Fix Series.max/min and Series.describe on len-0 series (GH1650)
- Handle None values in dict passed to concat (GH1649)
- Fix Series.interpolate with method='values' and DatetimeIndex (GH1646)
- Fix IndexError in left merges on a DataFrame with 0-length (GH1628)
- Fix DataFrame column width display with UTF-8 encoded characters (GH1620)
- Handle case in pandas.io.data.get\_data\_yahoo where Yahoo! returns duplicate dates for most recent business day
- Avoid downsampling when plotting mixed frequencies on the same subplot (GH1619)
- Fix read\_csv bug when reading a single line (GH1553)
- Fix bug in C code causing monthly periods prior to December 1969 to be off (GH1570)

## 38.32 pandas 0.8.0

**Release date:** 6/29/2012

### 38.32.1 New Features

- New unified DatetimeIndex class for nanosecond-level timestamp data
- New Timestamp datetime.datetime subclass with easy time zone conversions, and support for nanoseconds

- New `PeriodIndex` class for timespans, calendar logic, and `Period` scalar object
- High performance resampling of timestamp and period data. New *resample* method of all pandas data structures
- New frequency names plus shortcut string aliases like '15h', '1h30min'
- Time series string indexing shorthand ([GH222](#))
- Add `week`, `dayofyear` array and other timestamp array-valued field accessor functions to `DatetimeIndex`
- Add `GroupBy.prod` optimized aggregation function and 'prod' fast time series conversion method ([GH1018](#))
- Implement robust frequency inference function and *inferred\_freq* attribute on `DatetimeIndex` ([GH391](#))
- New `tz_convert` and `tz_localize` methods in `Series` / `DataFrame`
- Convert `DatetimeIndex`s to UTC if time zones are different in `join`/`setops` ([GH864](#))
- Add `limit` argument for forward/backward filling to `reindex`, `fillna`, etc. ([GH825](#) and others)
- Add support for indexes (dates or otherwise) with duplicates and common sense indexing/selection functionality
- `Series/DataFrame.update` methods, in-place variant of `combine_first` ([GH961](#))
- Add `match` function to API ([GH502](#))
- Add Cython-optimized `first`, `last`, `min`, `max`, `prod` functions to `GroupBy` ([GH994](#), [GH1043](#))
- Dates can be split across multiple columns ([GH1227](#), [GH1186](#))
- Add experimental support for converting pandas `DataFrame` to R `data.frame` via `rpy2` ([GH350](#), [GH1212](#))
- Can pass list of (name, function) to `GroupBy.aggregate` to get aggregates in a particular order ([GH610](#))
- Can pass dicts with lists of functions or dicts to `GroupBy.aggregate` to do much more flexible multiple function aggregation ([GH642](#), [GH610](#))
- New `ordered_merge` functions for merging `DataFrames` with ordered data. Also supports group-wise merging for panel data ([GH813](#))
- Add `keys()` method to `DataFrame`
- Add flexible `replace` method for replacing potentially values to `Series` and `DataFrame` ([GH929](#), [GH1241](#))
- Add 'kde' plot kind for `Series/DataFrame.plot` ([GH1059](#))
- More flexible multiple function aggregation with `GroupBy`
- Add `pct_change` function to `Series/DataFrame`
- Add option to interpolate by Index values in `Series.interpolate` ([GH1206](#))
- Add `max_colwidth` option for `DataFrame`, defaulting to 50
- Conversion of `DataFrame` through `rpy2` to R `data.frame` ([GH1282](#), )
- Add `keys()` method on `DataFrame` ([GH1240](#))
- Add new `match` function to API (similar to R) ([GH502](#))
- Add `dayfirst` option to parsers ([GH854](#))
- Add `method` argument to `align` method for forward/backward fillin ([GH216](#))
- Add `Panel.transpose` method for rearranging axes ([GH695](#))
- Add new `cut` function (patterned after R) for discretizing data into equal range-length bins or arbitrary breaks of your choosing ([GH415](#))
- Add new `qcut` for cutting with quantiles ([GH1378](#))



- Add `value_counts` top level array method ([GH1392](#))
- Added Andrews curves plot tupe ([GH1325](#))
- Add lag plot ([GH1440](#))
- Add autocorrelation\_plot ([GH1425](#))
- Add support for tox and Travis CI ([GH1382](#))
- Add support for Categorical use in GroupBy ([GH292](#))
- Add `any` and `all` methods to DataFrame ([GH1416](#))
- Add `secondary_y` option to Series.plot
- Add experimental `reshape` function for reshaping wide to long

### 38.32.2 Improvements to existing features

- Switch to klib/khash-based hash tables in Index classes for better performance in many cases and lower memory footprint
- Shipping some functions from `scipy.stats` to reduce dependency, e.g. `Series.describe` and `DataFrame.describe` ([GH1092](#))
- Can create MultiIndex by passing list of lists or list of arrays to Series, DataFrame constructor, etc. ([GH831](#))
- Can pass arrays in addition to column names to `DataFrame.set_index` ([GH402](#))
- Improve the speed of “square” reindexing of homogeneous DataFrame objects by significant margin ([GH836](#))
- Handle more dtypes when passed MaskedArrays in DataFrame constructor ([GH406](#))
- Improved performance of join operations on integer keys ([GH682](#))
- Can pass multiple columns to GroupBy object, e.g. `grouped[[col1, col2]]` to only aggregate a subset of the value columns ([GH383](#))
- Add histogram / kde plot options for `scatter_matrix` diagonals ([GH1237](#))
- Add inplace option to `Series/DataFrame.rename` and `sort_index`, `DataFrame.drop_duplicates` ([GH805](#), [GH207](#))
- More helpful error message when nothing passed to `Series.reindex` ([GH1267](#))
- Can mix array and scalars as dict-value inputs to DataFrame ctor ([GH1329](#))
- Use DataFrame columns’ name for legend title in plots
- Preserve frequency in DatetimeIndex when possible in boolean indexing operations
- Promote `datetime.date` values in data alignment operations ([GH867](#))
- Add `order` method to Index classes ([GH1028](#))
- Avoid hash table creation in large monotonic hash table indexes ([GH1160](#))
- Store time zones in HDFStore ([GH1232](#))
- Enable storage of sparse data structures in HDFStore ([GH85](#))
- Enable `Series.asof` to work with arrays of timestamp inputs
- Cython implementation of `DataFrame.corr` speeds up by > 100x ([GH1349](#), [GH1354](#))
- Exclude “nuisance” columns automatically in `GroupBy.transform` ([GH1364](#))
- Support functions-as-strings in `GroupBy.transform` ([GH1362](#))



- Use index name as xlabel/ylabel in plots ([GH1415](#))
- Add `convert_dtype` option to `Series.apply` to be able to leave data as `dtype=object` ([GH1414](#))
- Can specify all index level names in `concat` ([GH1419](#))
- Add `dialect` keyword to parsers for quoting conventions ([GH1363](#))
- Enable `DataFrame[bool_DataFrame] += value` ([GH1366](#))
- Add `retries` argument to `get_data_yahoo` to try to prevent Yahoo! API 404s ([GH826](#))
- Improve performance of reshaping by using  $O(N)$  categorical sorting
- Series names will be used for index of `DataFrame` if no index passed ([GH1494](#))
- Header argument in `DataFrame.to_csv` can accept a list of column names to use instead of the object's columns ([GH921](#))
- Add `raise_conflict` argument to `DataFrame.update` ([GH1526](#))
- Support file-like objects in `ExcelFile` ([GH1529](#))

### 38.32.3 API Changes

- Rename `pandas._tseries` to `pandas.lib`
- Rename `Factor` to `Categorical` and add improvements. Numerous `Categorical` bug fixes
- Frequency name overhaul, `WEEKDAY/EOM` and rules with `@` deprecated. `get_legacy_offset_name` backwards compatibility function added
- Raise `ValueError` in `DataFrame.__nonzero__`, so “if df” no longer works ([GH1073](#))
- Change `BDay` (business day) to not normalize dates by default ([GH506](#))
- Remove deprecated `DataMatrix` name
- Default merge suffixes for overlap now have underscores instead of periods to facilitate tab completion, etc. ([GH1239](#))
- Deprecation of `offset`, `time_rule` `timeRule` parameters throughout codebase
- `Series.append` and `DataFrame.append` no longer check for duplicate indexes by default, add `verify_integrity` parameter ([GH1394](#))
- Refactor `Factor` class, old constructor moved to `Factor.from_array`
- Modified internals of `MultiIndex` to use less memory (no longer represented as array of tuples) internally, speed up construction time and many methods which construct intermediate hierarchical indexes ([GH1467](#))

### 38.32.4 Bug Fixes

- Fix `OverflowError` from storing pre-1970 dates in `HDFStore` by switching to `datetime64` ([GH179](#))
- Fix logical error with February leap year end in `YearEnd` offset
- `Series([False, nan])` was getting casted to `float64` ([GH1074](#))
- Fix binary operations between boolean `Series` and object `Series` with booleans and NAs ([GH1074](#), [GH1079](#))
- Couldn't assign whole array to column in mixed-type `DataFrame` via `.ix` ([GH1142](#))
- Fix label slicing issues with float index values ([GH1167](#))

- Fix segfault caused by empty groups passed to groupby ([GH1048](#))
- Fix occasionally misbehaved reindexing in the presence of NaN labels ([GH522](#))
- Fix imprecise logic causing weird Series results from .apply ([GH1183](#))
- Unstack multiple levels in one shot, avoiding empty columns in some cases. Fix pivot table bug ([GH1181](#))
- Fix formatting of MultiIndex on Series/DataFrame when index name coincides with label ([GH1217](#))
- Handle Excel 2003 #N/A as NaN from xlrd ([GH1213](#), [GH1225](#))
- Fix timestamp locale-related deserialization issues with HDFStore by moving to datetime64 representation ([GH1081](#), [GH809](#))
- Fix DataFrame.duplicated/drop\_duplicates NA value handling ([GH557](#))
- Actually raise exceptions in fast reducer ([GH1243](#))
- Fix various timezone-handling bugs from 0.7.3 ([GH969](#))
- GroupBy on level=0 discarded index name ([GH1313](#))
- Better error message with unmergeable DataFrames ([GH1307](#))
- Series.\_\_repr\_\_ alignment fix with unicode index values ([GH1279](#))
- Better error message if nothing passed to reindex ([GH1267](#))
- More robust NA handling in DataFrame.drop\_duplicates ([GH557](#))
- Resolve locale-based and pre-epoch HDF5 timestamp deserialization issues ([GH973](#), [GH1081](#), [GH179](#))
- Implement Series.repeat ([GH1229](#))
- Fix indexing with namedtuple and other tuple subclasses ([GH1026](#))
- Fix float64 slicing bug ([GH1167](#))
- Parsing integers with commas ([GH796](#))
- Fix groupby improper data type when group consists of one value ([GH1065](#))
- Fix negative variance possibility in nanvar resulting from floating point error ([GH1090](#))
- Consistently set name on groupby pieces ([GH184](#))
- Treat dict return values as Series in GroupBy.apply ([GH823](#))
- Respect column selection for DataFrame in GroupBy.transform ([GH1365](#))
- Fix MultiIndex partial indexing bug ([GH1352](#))
- Enable assignment of rows in mixed-type DataFrame via .ix ([GH1432](#))
- Reset index mapping when grouping Series in Cython ([GH1423](#))
- Fix outer/inner DataFrame.join with non-unique indexes ([GH1421](#))
- Fix MultiIndex groupby bugs with empty lower levels ([GH1401](#))
- Calling fillna with a Series will have same behavior as with dict ([GH1486](#))
- SparseSeries reduction bug ([GH1375](#))
- Fix unicode serialization issue in HDFStore ([GH1361](#))
- Pass keywords to pyplot.boxplot in DataFrame.boxplot ([GH1493](#))
- Bug fixes in MonthBegin ([GH1483](#))

- Preserve MultiIndex names in drop ([GH1513](#))
- Fix Panel DataFrame slice-assignment bug ([GH1533](#))
- Don't use locals() in read\_\* functions ([GH1547](#))

## 38.33 pandas 0.7.3

**Release date:** April 12, 2012

### 38.33.1 New Features

- Support for non-unique indexes: indexing and selection, many-to-one and many-to-many joins ([GH1306](#))
- Added fixed-width file reader, read\_fwf ([GH952](#))
- Add group\_keys argument to groupby to not add group names to MultiIndex in result of apply ([GH938](#))
- DataFrame can now accept non-integer label slicing ([GH946](#)). Previously only DataFrame.ix was able to do so.
- DataFrame.apply now retains name attributes on Series objects ([GH983](#))
- Numeric DataFrame comparisons with non-numeric values now raises proper TypeError ([GH943](#)). Previously raise "PandasError: DataFrame constructor not properly called!"
- Add kurt methods to Series and DataFrame ([GH964](#))
- Can pass dict of column -> list/set NA values for text parsers ([GH754](#))
- Allows users specified NA values in text parsers ([GH754](#))
- Parsers checks for openpyxl dependency and raises ImportError if not found ([GH1007](#))
- New factory function to create HDFStore objects that can be used in a with statement so users do not have to explicitly call HDFStore.close ([GH1005](#))
- pivot\_table is now more flexible with same parameters as groupby ([GH941](#))
- Added stacked bar plots ([GH987](#))
- scatter\_matrix method in pandas/tools/plotting.py ([GH935](#))
- DataFrame.boxplot returns plot results for ex-post styling ([GH985](#))
- Short version number accessible as pandas.version.short\_version ([GH930](#))
- Additional documentation in panel.to\_frame ([GH942](#))
- More informative Series.apply docstring regarding element-wise apply ([GH977](#))
- Notes on rpy2 installation ([GH1006](#))
- Add rotation and font size options to hist method ([GH1012](#))
- Use exogenous / X variable index in result of OLS.y\_predict. Add OLS.predict method ([GH1027](#), [GH1008](#))

### 38.33.2 API Changes

- Calling apply on grouped Series, e.g. describe(), will no longer yield DataFrame by default. Will have to call unstack() to get prior behavior
- NA handling in non-numeric comparisons has been tightened up ([GH933](#), [GH953](#))

- No longer assign dummy names key\_0, key\_1, etc. to groupby index ([GH1291](#))

### 38.33.3 Bug Fixes

- Fix logic error when selecting part of a row in a DataFrame with a MultiIndex index ([GH1013](#))
- Series comparison with Series of differing length causes crash ([GH1016](#)).
- Fix bug in indexing when selecting section of hierarchically-indexed row ([GH1013](#))
- DataFrame.plot(logy=True) has no effect ([GH1011](#)).
- Broken arithmetic operations between SparsePanel-Panel ([GH1015](#))
- Unicode repr issues in MultiIndex with non-ASCII characters ([GH1010](#))
- DataFrame.lookup() returns inconsistent results if exact match not present ([GH1001](#))
- DataFrame arithmetic operations not treating None as NA ([GH992](#))
- DataFrameGroupBy.apply returns incorrect result ([GH991](#))
- Series.reshape returns incorrect result for multiple dimensions ([GH989](#))
- Series.std and Series.var ignores ddof parameter ([GH934](#))
- DataFrame.append loses index names ([GH980](#))
- DataFrame.plot(kind='bar') ignores color argument ([GH958](#))
- Inconsistent Index comparison results ([GH948](#))
- Improper int dtype DataFrame construction from data with NaN ([GH846](#))
- Removes default 'result' name in groupby results ([GH995](#))
- DataFrame.from\_records no longer mutate input columns ([GH975](#))
- Use Index name when grouping by it ([GH1313](#))

## 38.34 pandas 0.7.2

**Release date:** March 16, 2012

### 38.34.1 New Features

- Add additional tie-breaking methods in DataFrame.rank ([GH874](#))
- Add ascending parameter to rank in Series, DataFrame ([GH875](#))
- Add sort\_columns parameter to allow unsorted plots ([GH918](#))
- IPython tab completion on GroupBy objects

### 38.34.2 API Changes

- Series.sum returns 0 instead of NA when called on an empty series. Analogously for a DataFrame whose rows or columns are length 0 ([GH844](#))

### 38.34.3 Improvements to existing features

- Don't use groups dict in `Grouper.size` (GH860)
- Use `khash` for `Series.value_counts`, add `raw` function to `algorithms.py` (GH861)
- Enable column access via attributes on `GroupBy` (GH882)
- Enable setting existing columns (only) via attributes on `DataFrame`, `Panel` (GH883)
- Intercept `__builtin__.sum` in `groupby` (GH885)
- Can pass dict to `DataFrame.fillna` to use different values per column (GH661)
- Can select multiple hierarchical groups by passing list of values in `.ix` (GH134)
- Add `level` keyword to `drop` for dropping values from a level (GH159)
- Add `coerce_float` option on `DataFrame.from_records` (GH893)
- Raise exception if passed `date_parser` fails in `read_csv`
- Add `axis` option to `DataFrame.fillna` (GH174)
- Fixes to `Panel` to make it easier to subclass (GH888)

### 38.34.4 Bug Fixes

- Fix overflow-related bugs in `groupby` (GH850, GH851)
- Fix unhelpful error message in parsers (GH856)
- Better err msg for failed boolean slicing of dataframe (GH859)
- `Series.count` cannot accept a string (level name) in the level argument (GH869)
- Group index platform int check (GH870)
- `concat` on `axis=1` and `ignore_index=True` raises `TypeError` (GH871)
- Further unicode handling issues resolved (GH795)
- Fix failure in multiindex-based access in `Panel` (GH880)
- Fix `DataFrame` boolean slice assignment failure (GH881)
- Fix `combineAdd` `NotImplementedError` for `SparseDataFrame` (GH887)
- Fix `DataFrame.to_html` encoding and columns (GH890, GH891, GH909)
- Fix na-filling handling in mixed-type `DataFrame` (GH910)
- Fix to `DataFrame.set_value` with non-existent row/col (GH911)
- Fix malformed block in `groupby` when excluding nuisance columns (GH916)
- Fix inconsistent NA handling in `dtype=object` arrays (GH925)
- Fix missing center-of-mass computation in `ewmcov` (GH862)
- Don't raise exception when opening read-only HDF5 file (GH847)
- Fix possible out-of-bounds memory access in 0-length `Series` (GH917)

## 38.35 pandas 0.7.1

**Release date:** February 29, 2012

### 38.35.1 New Features

- Add `to_clipboard` function to pandas namespace for writing objects to the system clipboard ([GH774](#))
- Add `itertuples` method to `DataFrame` for iterating through the rows of a dataframe as tuples ([GH818](#))
- Add ability to pass `fill_value` and `method` to `DataFrame` and `Series` `align` method ([GH806](#), [GH807](#))
- Add `fill_value` option to `reindex`, `align` methods ([GH784](#))
- Enable `concat` to produce `DataFrame` from `Series` ([GH787](#))
- Add `between` method to `Series` ([GH802](#))
- Add HTML representation hook to `DataFrame` for the IPython HTML notebook ([GH773](#))
- Support for reading Excel 2007 XML documents using `openpyxl`

### 38.35.2 Improvements to existing features

- Improve performance and memory usage of `fillna` on `DataFrame`
- Can concatenate a list of `Series` along `axis=1` to obtain a `DataFrame` ([GH787](#))

### 38.35.3 Bug Fixes

- Fix memory leak when inserting large number of columns into a single `DataFrame` ([GH790](#))
- Appending length-0 `DataFrame` with new columns would not result in those new columns being part of the resulting concatenated `DataFrame` ([GH782](#))
- Fixed `groupby` corner case when passing dictionary grouper and `as_index` is `False` ([GH819](#))
- Fixed bug whereby bool array sometimes had object dtype ([GH820](#))
- Fix exception thrown on `np.diff` ([GH816](#))
- Fix `to_records` where columns are non-strings ([GH822](#))
- Fix `Index.intersection` where indices have incomparable types ([GH811](#))
- Fix `ExcelFile` throwing an exception for two-line file ([GH837](#))
- Add clearer error message in csv parser ([GH835](#))
- Fix loss of fractional seconds in `HDFStore` ([GH513](#))
- Fix `DataFrame` join where columns have datetimes ([GH787](#))
- Work around NumPy performance issue in `take` ([GH817](#))
- Improve comparison operations for NA-friendliness ([GH801](#))
- Fix indexing operation for floating point values ([GH780](#), [GH798](#))
- Fix `groupby` case resulting in malformed dataframe ([GH814](#))
- Fix behavior of `reindex` of `Series` dropping name ([GH812](#))

- Improve on redundant groupby computation ([GH775](#))
- Catch possible NA assignment to int/bool series with exception ([GH839](#))

## 38.36 pandas 0.7.0

Release date: 2/9/2012

### 38.36.1 New Features

- New `merge` function for efficiently performing full gamut of database / relational-algebra operations. Refactored existing join methods to use the new infrastructure, resulting in substantial performance gains ([GH220](#), [GH249](#), [GH267](#))
- New `concat` function for concatenating `DataFrame` or `Panel` objects along an axis. Can form union or intersection of the other axes. Improves performance of `DataFrame.append` ([GH468](#), [GH479](#), [GH273](#))
- Handle differently-indexed output values in `DataFrame.apply` ([GH498](#))
- Can pass list of dicts (e.g., a list of shallow JSON objects) to `DataFrame` constructor ([GH526](#))
- Add `reorder_levels` method to `Series` and `DataFrame` ([GH534](#))
- Add dict-like `get` function to `DataFrame` and `Panel` ([GH521](#))
- `DataFrame.iterrows` method for efficiently iterating through the rows of a `DataFrame`
- Added `DataFrame.to_panel` with code adapted from `LongPanel.to_long`
- `reindex_axis` method added to `DataFrame`
- Add `level` option to binary arithmetic functions on `DataFrame` and `Series`
- Add `level` option to the `reindex` and `align` methods on `Series` and `DataFrame` for broadcasting values across a level ([GH542](#), [GH552](#), others)
- Add attribute-based item access to `Panel` and add IPython completion (PR [GH554](#))
- Add `logy` option to `Series.plot` for log-scaling on the Y axis
- Add `index`, `header`, and `justify` options to `DataFrame.to_string`. Add option to ([GH570](#), [GH571](#))
- Can pass multiple `DataFrames` to `DataFrame.join` to join on index ([GH115](#))
- Can pass multiple `Panels` to `Panel.join` ([GH115](#))
- Can pass multiple `DataFrames` to `DataFrame.append` to concatenate (stack) and multiple `Series` to `Series.append` too
- Added `justify` argument to `DataFrame.to_string` to allow different alignment of column headers
- Add `sort` option to `GroupBy` to allow disabling sorting of the group keys for potential speedups ([GH595](#))
- Can pass `MaskedArray` to `Series` constructor ([GH563](#))
- Add `Panel` item access via attributes and IPython completion ([GH554](#))
- Implement `DataFrame.lookup`, fancy-indexing analogue for retrieving values given a sequence of row and column labels ([GH338](#))
- Add `verbose` option to `read_csv` and `read_table` to show number of NA values inserted in non-numeric columns ([GH614](#))

- Can pass a list of dicts or Series to `DataFrame.append` to concatenate multiple rows (GH464)
- Add `level` argument to `DataFrame.xs` for selecting data from other MultiIndex levels. Can take one or more levels with potentially a tuple of keys for flexible retrieval of data (GH371, GH629)
- New `crosstab` function for easily computing frequency tables (GH170)
- Can pass a list of functions to aggregate with `groupby` on a `DataFrame`, yielding an aggregated result with hierarchical columns (GH166)
- Add integer-indexing functions `iget` in `Series` and `irow / iget` in `DataFrame` (GH628)
- Add new `Series.unique` function, significantly faster than `numpy.unique` (GH658)
- Add new `cummin` and `cummax` instance methods to `Series` and `DataFrame` (GH647)
- Add new `value_range` function to return min/max of a dataframe (GH288)
- Add `drop` parameter to `reset_index` method of `DataFrame` and added method to `Series` as well (GH699)
- Add `isin` method to `Index` objects, works just like `Series.isin` (GH GH657)
- Implement array interface on `Panel` so that ufuncs work (re: GH740)
- Add `sort` option to `DataFrame.join` (GH731)
- Improved handling of NAs (propagation) in binary operations with `dtype=object` arrays (GH737)
- Add `abs` method to `Pandas` objects
- Added `algorithms` module to start collecting central algos

### 38.36.2 API Changes

- Label-indexing with integer indexes now raises `KeyError` if a label is not found instead of falling back on location-based indexing (GH700)
- Label-based slicing via `ix` or `[]` on `Series` will now only work if exact matches for the labels are found or if the index is monotonic (for range selections)
- Label-based slicing and sequences of labels can be passed to `[]` on a `Series` for both getting and setting (GH86)
- `[]` operator (`__getitem__` and `__setitem__`) will raise `KeyError` with integer indexes when an index is not contained in the index. The prior behavior would fall back on position-based indexing if a key was not found in the index which would lead to subtle bugs. This is now consistent with the behavior of `.ix` on `DataFrame` and friends (GH328)
- Rename `DataFrame.delevel` to `DataFrame.reset_index` and add deprecation warning
- `Series.sort` (an in-place operation) called on a `Series` which is a view on a larger array (e.g. a column in a `DataFrame`) will generate an `Exception` to prevent accidentally modifying the data source (GH316)
- Refactor to remove deprecated `LongPanel` class (GH552)
- Deprecated `Panel.to_long`, renamed to `to_frame`
- Deprecated `colSpace` argument in `DataFrame.to_string`, renamed to `col_space`
- Rename `precision` to `accuracy` in engineering float formatter (GH GH395)
- The default delimiter for `read_csv` is comma rather than letting `csv.Sniffer` infer it
- Rename `col_or_columns` argument in `DataFrame.drop_duplicates` (GH GH734)



### 38.36.3 Improvements to existing features

- Better error message in DataFrame constructor when passed column labels don't match data ([GH497](#))
- Substantially improve performance of multi-GroupBy aggregation when a Python function is passed, reuse ndarray object in Cython ([GH496](#))
- Can store objects indexed by tuples and floats in HDFStore ([GH492](#))
- Don't print length by default in Series.to\_string, add *length* option ([GH](#) [GH489](#))
- Improve Cython code for multi-groupby to aggregate without having to sort the data ([GH93](#))
- Improve MultiIndex reindexing speed by storing tuples in the MultiIndex, test for backwards unpickling compatibility
- Improve column reindexing performance by using specialized Cython take function
- Further performance tweaking of Series.\_\_getitem\_\_ for standard use cases
- Avoid Index dict creation in some cases (i.e. when getting slices, etc.), regression from prior versions
- Friendlier error message in setup.py if NumPy not installed
- Use common set of NA-handling operations (sum, mean, etc.) in Panel class also ([GH536](#))
- Default name assignment when calling reset\_index on DataFrame with a regular (non-hierarchical) index ([GH476](#))
- Use Cythonized groupers when possible in Series/DataFrame stat ops with level parameter passed ([GH545](#))
- Ported skiplist data structure to C to speed up rolling\_median by about 5-10x in most typical use cases ([GH374](#))
- Some performance enhancements in constructing a Panel from a dict of DataFrame objects
- Made Index.\_get\_duplicates a public method by removing the underscore
- Prettier printing of floats, and column spacing fix ([GH395](#), [GH571](#))
- Add bold\_rows option to DataFrame.to\_html ([GH586](#))
- Improve the performance of DataFrame.sort\_index by up to 5x or more when sorting by multiple columns
- Substantially improve performance of DataFrame and Series constructors when passed a nested dict or dict, respectively ([GH540](#), [GH621](#))
- Modified setup.py so that pip / setuptools will install dependencies ([GH](#) [GH507](#), various pull requests)
- Unstack called on DataFrame with non-MultiIndex will return Series ([GH](#) [GH477](#))
- Improve DataFrame.to\_string and console formatting to be more consistent in the number of displayed digits ([GH395](#))
- Use bottleneck if available for performing NaN-friendly statistical operations that it implemented ([GH91](#))
- Monkey-patch context to traceback in DataFrame.apply to indicate which row/column the function application failed on ([GH614](#))
- Improved ability of read\_table and read\_clipboard to parse console-formatted DataFrames (can read the row of index names, etc.)
- Can pass list of group labels (without having to convert to an ndarray yourself) to groupby in some cases ([GH659](#))
- Use kind argument to Series.order for selecting different sort kinds ([GH668](#))

- Add option to `Series.to_csv` to omit the index ([GH684](#))
- Add `delimiter` as an alternative to `sep` in `read_csv` and other parsing functions
- Substantially improved performance of `groupby` on DataFrames with many columns by aggregating blocks of columns all at once ([GH745](#))
- Can pass a file handle or `StringIO` to `Series/DataFrame.to_csv` ([GH765](#))
- Can pass sequence of integers to `DataFrame.irow(icol)` and `Series.iget`, ([GH GH654](#))
- Prototypes for some vectorized string functions
- Add float64 hash table to solve the `Series.unique` problem with NAs ([GH714](#))
- Memoize objects when reading from file to reduce memory footprint
- Can get and set a column of a DataFrame with hierarchical columns containing “empty” (“”) lower levels without passing the empty levels ([PR GH768](#))

### 38.36.4 Bug Fixes

- Raise exception in out-of-bounds indexing of `Series` instead of seg-faulting, regression from earlier releases ([GH495](#))
- Fix error when joining DataFrames of different dtypes within the same typeclass (e.g. `float32` and `float64`) ([GH486](#))
- Fix bug in `Series.min/Series.max` on objects like `datetime.datetime` ([GH GH487](#))
- Preserve index names in `Index.union` ([GH501](#))
- Fix bug in `Index` joining causing subclass information (like `DateRange` type) to be lost in some cases ([GH500](#))
- Accept empty list as input to `DataFrame` constructor, regression from 0.6.0 ([GH491](#))
- Can output `DataFrame` and `Series` with `ndarray` objects in a `dtype=object` array ([GH490](#))
- Return empty string from `Series.to_string` when called on empty `Series` ([GH GH488](#))
- Fix exception passing empty list to `DataFrame.from_records`
- Fix `Index.format` bug (excluding name field) with datetimes with time info
- Fix scalar value access in `Series` to always return NumPy scalars, regression from prior versions ([GH510](#))
- Handle rows skipped at beginning of file in `read_*` functions ([GH505](#))
- Handle improper dtype casting in `set_value` methods
- Unary `'-'` / `__neg__` operator on `DataFrame` was returning integer values
- Unbox 0-dim `ndarrays` from certain operators like `all`, `any` in `Series`
- Fix handling of missing columns (was `combine_first`-specific) in `DataFrame.combine` for general case ([GH529](#))
- Fix type inference logic with boolean lists and arrays in `DataFrame` indexing
- Use centered sum of squares in R-square computation if `entity_effects=True` in panel regression
- Handle all NA case in `Series.{corr, cov}`, was raising exception ([GH548](#))
- Aggregating by multiple levels with `level` argument to `DataFrame`, `Series` stat method, was broken ([GH545](#))
- Fix Cython buf when converter passed to `read_csv` produced a numeric array (buffer dtype mismatch when passed to Cython type inference function) ([GH GH546](#))
- Fix exception when setting scalar value using `.ix` on a `DataFrame` with a `MultiIndex` ([GH551](#))

- Fix outer join between two DateRanges with different offsets that returned an invalid DateRange
- Cleanup DataFrame.from\_records failure where index argument is an integer
- Fix Data.from\_records failure when passed a dictionary
- Fix NA handling in {Series, DataFrame}.rank with non-floating point dtypes
- Fix bug related to integer type-checking in .ix-based indexing
- Handle non-string index name passed to DataFrame.from\_records
- DataFrame.insert caused the columns name(s) field to be discarded (GH527)
- Fix erroneous in monotonic many-to-one left joins
- Fix DataFrame.to\_string to remove extra column white space (GH571)
- Format floats to default to same number of digits (GH395)
- Added decorator to copy docstring from one function to another (GH449)
- Fix error in monotonic many-to-one left joins
- Fix \_\_eq\_\_ comparison between DateOffsets with different relativedelta keywords passed
- Fix exception caused by parser converter returning strings (GH583)
- Fix MultiIndex formatting bug with integer names (GH601)
- Fix bug in handling of non-numeric aggregates in Series.groupby (GH612)
- Fix TypeError with tuple subclasses (e.g. namedtuple) in DataFrame.from\_records (GH611)
- Catch misreported console size when running IPython within Emacs
- Fix minor bug in pivot table margins, loss of index names and length-1 'All' tuple in row labels
- Add support for legacy WidePanel objects to be read from HDFStore
- Fix out-of-bounds segfault in pad\_object and backfill\_object methods when either source or target array are empty
- Could not create a new column in a DataFrame from a list of tuples
- Fix bugs preventing SparseDataFrame and SparseSeries working with groupby (GH666)
- Use sort kind in Series.sort / argsort (GH668)
- Fix DataFrame operations on non-scalar, non-pandas objects (GH672)
- Don't convert DataFrame column to integer type when passing integer to \_\_setitem\_\_ (GH669)
- Fix downstream bug in pivot\_table caused by integer level names in MultiIndex (GH678)
- Fix SparseSeries.combine\_first when passed a dense Series (GH687)
- Fix performance regression in HDFStore loading when DataFrame or Panel stored in table format with datetimes
- Raise Exception in DateRange when offset with n=0 is passed (GH683)
- Fix get/set inconsistency with .ix property and integer location but non-integer index (GH707)
- Use right dropna function for SparseSeries. Return dense Series for NA fill value (GH730)
- Fix Index.format bug causing incorrectly string-formatted Series with datetime indexes (GH726, GH758)
- Fix errors caused by object dtype arrays passed to ols (GH759)
- Fix error where column names lost when passing list of labels to DataFrame.\_\_getitem\_\_, (GH662)

- Fix error whereby top-level week iterator overwrote week instance
- Fix circular reference causing memory leak in sparse array / series / frame, ([GH663](#))
- Fix integer-slicing from integers-as-floats ([GH670](#))
- Fix zero division errors in nanops from object dtype arrays in all NA case ([GH676](#))
- Fix csv encoding when using unicode ([GH705](#), [GH717](#), [GH738](#))
- Fix assumption that each object contains every unique block type in concat, ([GH708](#))
- Fix sortedness check of multiindex in to\_panel ([GH719](#), 720)
- Fix that None was not treated as NA in PyObjectHashtable
- Fix hashing dtype because of endianness confusion ([GH747](#), [GH748](#))
- Fix SparseSeries.dropna to return dense Series in case of NA fill value (GH [GH730](#))
- Use map\_infer instead of np.vectorize. handle NA sentinels if converter yields numeric array, ([GH753](#))
- Fixes and improvements to DataFrame.rank ([GH742](#))
- Fix catching AttributeError instead of NameError for bottleneck
- Try to cast non-MultiIndex to better dtype when calling reset\_index ([GH726](#) [GH440](#))
- Fix #1.QNAN0' float bug on 2.6/win64
- Allow subclasses of dicts in DataFrame constructor, with tests
- Fix problem whereby set\_index destroys column multiindex ([GH764](#))
- Hack around bug in generating DateRange from naive DateOffset ([GH770](#))
- Fix bug in DateRange.intersection causing incorrect results with some overlapping ranges ([GH771](#))

### 38.36.5 Thanks

- Craig Austin
- Chris Billington
- Marius Cobzarencu
- Mario Gamboa-Cavazos
- Hans-Martin Gaudecker
- Arthur Gerigk
- Yaroslav Halchenko
- Jeff Hammerbacher
- Matt Harrison
- Andreas Hilboll
- Luc Kesters
- Adam Klein
- Gregg Lind
- Solomon Negusse
- Wouter Overmeire

- Christian Prinoth
- Jeff Reback
- Sam Reckoner
- Craig Reeson
- Jan Schulz
- Skipper Seabold
- Ted Square
- Graham Taylor
- Aman Thakral
- Chris Uga
- Dieter Vandenbussche
- Texas P.
- Pinxing Ye
- ... and everyone I forgot

## 38.37 pandas 0.6.1

**Release date:** 12/13/2011

### 38.37.1 API Changes

- Rename *names* argument in `DataFrame.from_records` to *columns*. Add deprecation warning
- Boolean get/set operations on Series with boolean Series will reindex instead of requiring that the indexes be exactly equal ([GH429](#))

### 38.37.2 New Features

- Can pass Series to `DataFrame.append` with `ignore_index=True` for appending a single row ([GH430](#))
- Add Spearman and Kendall correlation options to `Series.corr` and `DataFrame.corr` ([GH428](#))
- Add new *get\_value* and *set\_value* methods to Series, DataFrame, and Panel to very low-overhead access to scalar elements. `df.get_value(row, column)` is about 3x faster than `df[column][row]` by handling fewer cases ([GH437](#), [GH438](#)). Add similar methods to sparse data structures for compatibility
- Add Qt table widget to sandbox ([GH435](#))
- `DataFrame.align` can accept Series arguments, add `axis` keyword ([GH461](#))
- Implement new `SparseList` and `SparseArray` data structures. `SparseSeries` now derives from `SparseArray` ([GH463](#))
- `max_columns` / `max_rows` options in `set_printoptions` ([GH453](#))
- Implement `Series.rank` and `DataFrame.rank`, fast versions of `scipy.stats.rankdata` ([GH428](#))
- Implement `DataFrame.from_items` alternate constructor ([GH444](#))

- `DataFrame.convert_objects` method for inferring better dtypes for object columns ([GH302](#))
- Add `rolling_corr_pairwise` function for computing Panel of correlation matrices ([GH189](#))
- Add `margins` option to `pivot_table` for computing subgroup aggregates ([GH114](#))
- Add `Series.from_csv` function ([GH482](#))

### 38.37.3 Improvements to existing features

- Improve memory usage of `DataFrame.describe` (do not copy data unnecessarily) ([GH425](#))
- Use same formatting function for outputting floating point Series to console as in DataFrame ([GH420](#))
- `DataFrame.delevel` will try to infer better dtype for new columns ([GH440](#))
- Exclude non-numeric types in `DataFrame.{corr, cov}`
- Override `Index.astype` to enable dtype casting ([GH412](#))
- Use same float formatting function for `Series.__repr__` ([GH420](#))
- Use available console width to output DataFrame columns ([GH453](#))
- Accept ndarrays when setting items in Panel ([GH452](#))
- Infer console width when printing `__repr__` of DataFrame to console (PR [GH453](#))
- Optimize scalar value lookups in the general case by 25% or more in Series and DataFrame
- Can pass DataFrame/DataFrame and DataFrame/Series to `rolling_corr/rolling_cov` ([GH462](#))
- Fix performance regression in cross-sectional count in DataFrame, affecting `DataFrame.dropna` speed
- Column deletion in DataFrame copies no data (computes views on blocks) ([GH158](#))
- `MultiIndex.get_level_values` can take the level name
- More helpful error message when `DataFrame.plot` fails on one of the columns ([GH478](#))
- Improve performance of `DataFrame.{index, columns}` attribute lookup

### 38.37.4 Bug Fixes

- Fix  $O(K^2)$  memory leak caused by inserting many columns without consolidating, had been present since 0.4.0 ([GH467](#))
- `DataFrame.count` should return Series with zero instead of NA with length-0 axis ([GH423](#))
- Fix Yahoo! Finance API usage in `pandas.io.data` ([GH419](#), [GH427](#))
- Fix upstream bug causing failure in `Series.align` with empty Series ([GH434](#))
- Function passed to `DataFrame.apply` can return a list, as long as it's the right length. Regression from 0.4 ([GH432](#))
- Don't "accidentally" upcast scalar values when indexing using `.ix` ([GH431](#))
- Fix groupby exception raised with `as_index=False` and single column selected ([GH421](#))
- Implement `DateOffset.__ne__` causing downstream bug ([GH456](#))
- Fix `__doc__`-related issue when converting py -> pyo with `py2exe`
- Bug fix in left join Cython code with duplicate monotonic labels

- Fix bug when unstacking multiple levels described in [GH451](#)
- Exclude NA values in dtype=object arrays, regression from 0.5.0 ([GH469](#))
- Use Cython map\_infer function in DataFrame.applymap to properly infer output type, handle tuple return values and other things that were breaking ([GH465](#))
- Handle floating point index values in HDFStore ([GH454](#))
- Fixed stale column reference bug (cached Series object) caused by type change / item deletion in DataFrame ([GH473](#))
- Index.get\_loc should always raise Exception when there are duplicates
- Handle differently-indexed Series input to DataFrame constructor ([GH475](#))
- Omit nuisance columns in multi-groupby with Python function
- Buglet in handling of single grouping in general apply
- Handle type inference properly when passing list of lists or tuples to DataFrame constructor ([GH484](#))
- Preserve Index / MultiIndex names in GroupBy.apply concatenation step (GH [GH481](#))

### 38.37.5 Thanks

- Ralph Bean
- Luca Beltrame
- Marius Cobzarencu
- Andreas Hilboll
- Jev Kuznetsov
- Adam Lichtenstein
- Wouter Overmeire
- Fernando Perez
- Nathan Pinger
- Christian Prinoth
- Alex Reyfman
- Joon Ro
- Chang She
- Ted Square
- Chris Uga
- Dieter Vandenbussche

### 38.38 pandas 0.6.0

**Release date:** 11/25/2011

### 38.38.1 API Changes

- Arithmetic methods like *sum* will attempt to sum dtype=object values by default instead of excluding them (GH382)

### 38.38.2 New Features

- Add *melt* function to *pandas.core.reshape*
- Add *level* parameter to group by level in Series and DataFrame descriptive statistics (GH313)
- Add *head* and *tail* methods to Series, analogous to DataFrame (PR GH296)
- Add *Series.isin* function which checks if each value is contained in a passed sequence (GH289)
- Add *float\_format* option to *Series.to\_string*
- Add *skip\_footer* (GH291) and *converters* (GH343) options to *read\_csv* and *read\_table*
- Add proper, tested weighted least squares to standard and panel OLS (GH GH303)
- Add *drop\_duplicates* and *duplicated* functions for removing duplicate DataFrame rows and checking for duplicate rows, respectively (GH319)
- Implement logical (boolean) operators *&*, *|*, *^* on DataFrame (GH347)
- Add *Series.mad*, mean absolute deviation, matching DataFrame
- Add *QuarterEnd* DateOffset (GH321)
- Add matrix multiplication function *dot* to DataFrame (GH65)
- Add *orient* option to *Panel.from\_dict* to ease creation of mixed-type Panels (GH359, GH301)
- Add *DataFrame.from\_dict* with similar *orient* option
- Can now pass list of tuples or list of lists to *DataFrame.from\_records* for fast conversion to DataFrame (GH357)
- Can pass multiple levels to groupby, e.g. *df.groupby(level=[0, 1])* (GH GH103)
- Can sort by multiple columns in *DataFrame.sort\_index* (GH92, GH362)
- Add fast *get\_value* and *put\_value* methods to DataFrame and micro-performance tweaks (GH360)
- Add *cov* instance methods to Series and DataFrame (GH194, GH362)
- Add bar plot option to *DataFrame.plot* (GH348)
- Add *idxmin* and *idxmax* functions to Series and DataFrame for computing index labels achieving maximum and minimum values (GH286)
- Add *read\_clipboard* function for parsing DataFrame from OS clipboard, should work across platforms (GH300)
- Add *nunique* function to Series for counting unique elements (GH297)
- DataFrame constructor will use Series name if no columns passed (GH373)
- Support regular expressions and longer delimiters in *read\_table/read\_csv*, but does not handle quoted strings yet (GH364)
- Add *DataFrame.to\_html* for formatting DataFrame to HTML (GH387)
- MaskedArray can be passed to DataFrame constructor and masked values will be converted to NaN (GH396)
- Add *DataFrame.boxplot* function (GH368, others)
- Can pass extra args, kwds to DataFrame.apply (GH376)



### 38.38.3 Improvements to existing features

- Raise more helpful exception if date parsing fails in `DateRange` (GH298)
- Vastly improved performance of `GroupBy` on axes with a `MultiIndex` (GH299)
- Print level names in hierarchical index in `Series` repr (GH305)
- Return `DataFrame` when performing `GroupBy` on selected column and `as_index=False` (GH308)
- Can pass vector to `on` argument in `DataFrame.join` (GH312)
- Don't show `Series` name if it's `None` in the repr, also omit length for short `Series` (GH317)
- Show legend by default in `DataFrame.plot`, add `legend` boolean flag (GH GH324)
- Significantly improved performance of `Series.order`, which also makes `np.unique` called on a `Series` faster (GH327)
- Faster cythonized count by level in `Series` and `DataFrame` (GH341)
- Raise exception if `dateutil` 2.0 installed on Python 2.x runtime (GH346)
- Significant `GroupBy` performance enhancement with multiple keys with many “empty” combinations
- New Cython vectorized function `map_infer` speeds up `Series.apply` and `Series.map` significantly when passed elementwise Python function, motivated by GH355
- Cythonized `cache_readonly`, resulting in substantial micro-performance enhancements throughout the codebase (GH361)
- Special Cython matrix iterator for applying arbitrary reduction operations with 3-5x better performance than `np.apply_along_axis` (GH309)
- Add `raw` option to `DataFrame.apply` for getting better performance when the passed function only requires an `ndarray` (GH309)
- Improve performance of `MultiIndex.from_tuples`
- Can pass multiple levels to `stack` and `unstack` (GH370)
- Can pass multiple values columns to `pivot_table` (GH381)
- Can call `DataFrame.delevel` with standard `Index` with name set (GH393)
- Use `Series` name in `GroupBy` for result index (GH363)
- Refactor `Series/DataFrame` stat methods to use common set of NaN-friendly function
- Handle NumPy scalar integers at C level in Cython conversion routines

### 38.38.4 Bug Fixes

- Fix bug in `DataFrame.to_csv` when writing a `DataFrame` with an index name (GH290)
- `DataFrame` should clear its `Series` caches on consolidation, was causing “stale” `Series` to be returned in some corner cases (GH304)
- `DataFrame` constructor failed if a column had a list of tuples (GH293)
- Ensure that `Series.apply` always returns a `Series` and implement `Series.round` (GH314)
- Support boolean columns in Cythonized groupby functions (GH315)
- `DataFrame.describe` should not fail if there are no numeric columns, instead return categorical describe (GH323)

- Fixed bug which could cause columns to be printed in wrong order in *DataFrame.to\_string* if specific list of columns passed (GH325)
- Fix legend plotting failure if DataFrame columns are integers (GH326)
- Shift start date back by one month for Yahoo! Finance API in *pandas.io.data* (GH329)
- Fix *DataFrame.join* failure on unconsolidated inputs (GH331)
- *DataFrame.min/max* will no longer fail on mixed-type DataFrame (GH337)
- Fix *read\_csv / read\_table* failure when passing list to *index\_col* that is not in ascending order (GH349)
- Fix failure passing *Int64Index* to *Index.union* when both are monotonic
- Fix error when passing *SparseSeries* to (dense) DataFrame constructor
- Added missing bang at top of *setup.py* (GH352)
- Change *is\_monotonic* on *MultiIndex* so it properly compares the tuples
- Fix *MultiIndex* outer join logic (GH351)
- Set index name attribute with single-key groupby (GH358)
- Bug fix in reflexive binary addition in Series and DataFrame for non-commutative operations (like string concatenation) (GH353)
- *setuptools.py* will invoke Cython (GH192)
- Fix block consolidation bug after inserting column into *MultiIndex* (GH366)
- Fix bug in join operations between *Index* and *Int64Index* (GH367)
- Handle *min\_periods=0* case in moving window functions (GH365)
- Fixed corner cases in *DataFrame.apply/pivot* with empty DataFrame (GH378)
- Fixed repr exception when Series name is a tuple
- Always return *DateRange* from *asfreq* (GH390)
- Pass level names to *swaplevel* (GH379)
- Don't lose index names in *MultiIndex.droplevel* (GH394)
- Infer more proper return type in *DataFrame.apply* when no columns or rows depending on whether the passed function is a reduction (GH389)
- Always return NA/NaN from *Series.min/max* and *DataFrame.min/max* when all of a row/column/values are NA (GH384)
- Enable partial setting with *.ix / advanced indexing* (GH397)
- Handle mixed-type DataFrames correctly in *unstack*, do not lose type information (GH403)
- Fix integer name formatting bug in *Index.format* and in *Series.\_\_repr\_\_*
- Handle label types other than string passed to groupby (GH405)
- Fix bug in *.ix*-based indexing with partial retrieval when a label is not contained in a level
- Index name was not being pickled (GH408)
- Level name should be passed to result index in *GroupBy.apply* (GH416)

### 38.38.5 Thanks

- Craig Austin
- Marius Cobzarencu
- Joel Cross
- Jeff Hammerbacher
- Adam Klein
- Thomas Kluyver
- Jev Kuznetsov
- Kieran O'Mahony
- Wouter Overmeire
- Nathan Pinger
- Christian Prinoth
- Skipper Seabold
- Chang She
- Ted Square
- Aman Thakral
- Chris Uga
- Dieter Vandenbussche
- carljv
- rsamson

## 38.39 pandas 0.5.0

**Release date:** 10/24/2011

This release of pandas includes a number of API changes (see below) and cleanup of deprecated APIs from pre-0.4.0 releases. There are also bug fixes, new features, numerous significant performance enhancements, and includes a new ipython completer hook to enable tab completion of DataFrame columns accesses and attributes (a new feature).

In addition to the changes listed here from 0.4.3 to 0.5.0, the minor releases 4.1, 0.4.2, and 0.4.3 brought some significant new functionality and performance improvements that are worth taking a look at.

Thanks to all for bug reports, contributed patches and generally providing feedback on the library.

### 38.39.1 API Changes

- *read\_table*, *read\_csv*, and *ExcelFile.parse* default arguments for *index\_col* is now None. To use one or more of the columns as the resulting DataFrame's index, these must be explicitly specified now
- Parsing functions like *read\_csv* no longer parse dates by default (GH [GH225](#))
- Removed *weights* option in panel regression which was not doing anything principled (GH155)
- Changed *buffer* argument name in *Series.to\_string* to *buf*

- *Series.to\_string* and *DataFrame.to\_string* now return strings by default instead of printing to `sys.stdout`
- Deprecated *nanRep* argument in various *to\_string* and *to\_csv* functions in favor of *na\_rep*. Will be removed in 0.6 ([GH275](#))
- Renamed *delimiter* to *sep* in *DataFrame.from\_csv* for consistency
- Changed order of *Series.clip* arguments to match those of *numpy.clip* and added (unimplemented) *out* argument so *numpy.clip* can be called on a Series ([GH272](#))
- Series functions renamed (and thus deprecated) in 0.4 series have been removed:
  - *asOf*, use *asof*
  - *toDict*, use *to\_dict*
  - *toString*, use *to\_string*
  - *toCSV*, use *to\_csv*
  - *merge*, use *map*
  - *applymap*, use *apply*
  - *combineFirst*, use *combine\_first*
  - *\_firstTimeWithValue* use *first\_valid\_index*
  - *\_lastTimeWithValue* use *last\_valid\_index*
- DataFrame functions renamed / deprecated in 0.4 series have been removed:
  - *asMatrix* method, use *as\_matrix* or *values* attribute
  - *combineFirst*, use *combine\_first*
  - *getXS*, use *xs*
  - *merge*, use *join*
  - *fromRecords*, use *from\_records*
  - *fromcsv*, use *from\_csv*
  - *toRecords*, use *to\_records*
  - *toDict*, use *to\_dict*
  - *toString*, use *to\_string*
  - *toCSV*, use *to\_csv*
  - *\_firstTimeWithValue* use *first\_valid\_index*
  - *\_lastTimeWithValue* use *last\_valid\_index*
  - *toDataMatrix* is no longer needed
  - *rows()* method, use *index* attribute
  - *cols()* method, use *columns* attribute
  - *dropEmptyRows()*, use *dropna(how='all')*
  - *dropIncompleteRows()*, use *dropna()*
  - *tapply(f)*, use *apply(f, axis=1)*
  - *tgroupby(keyfunc, aggfunc)*, use *groupby* with *axis=1*

### 38.39.2 Deprecations Removed

- *indexField* argument in *DataFrame.from\_records*
- *missingAtEnd* argument in *Series.order*. Use *na\_last* instead
- *Series.fromValue* classmethod, use regular *Series* constructor instead
- Functions *parseCSV*, *parseText*, and *parseExcel* methods in *pandas.io.parsers* have been removed
- *Index.asOfDate* function
- *Panel.getMinorXS* (use *minor\_xs*) and *Panel.getMajorXS* (use *major\_xs*)
- *Panel.toWide*, use *Panel.to\_wide* instead

### 38.39.3 New Features

- Added *DataFrame.align* method with standard join options
- Added *parse\_dates* option to *read\_csv* and *read\_table* methods to optionally try to parse dates in the index columns
- Add *nrows*, *chunksize*, and *iterator* arguments to *read\_csv* and *read\_table*. The last two return a new *TextParser* class capable of lazily iterating through chunks of a flat file (GH242)
- Added ability to join on multiple columns in *DataFrame.join* (GH214)
- Added private *\_get\_duplicates* function to *Index* for identifying duplicate values more easily
- Added column attribute access to *DataFrame*, e.g. *df.A* equivalent to *df['A']* if 'A' is a column in the *DataFrame* (GH213)
- Added IPython tab completion hook for *DataFrame* columns. (GH233, GH230)
- Implement *Series.describe* for *Series* containing objects (GH241)
- Add inner join option to *DataFrame.join* when joining on key(s) (GH248)
- Can select set of *DataFrame* columns by passing a list to *\_\_getitem\_\_* (GH GH253)
- Can use *&* and *|* to intersection / union *Index* objects, respectively (GH GH261)
- Added *pivot\_table* convenience function to pandas namespace (GH234)
- Implemented *Panel.rename\_axis* function (GH243)
- *DataFrame* will show index level names in console output
- Implemented *Panel.take*
- Add *set\_eng\_float\_format* function for setting alternate *DataFrame* floating point string formatting
- Add convenience *set\_index* function for creating a *DataFrame* index from its existing columns

### 38.39.4 Improvements to existing features

- Major performance improvements in file parsing functions *read\_csv* and *read\_table*
- Added Cython function for converting tuples to ndarray very fast. Speeds up many MultiIndex-related operations
- File parsing functions like *read\_csv* and *read\_table* will explicitly check if a parsed index has duplicates and raise a more helpful exception rather than deferring the check until later

- Refactored merging / joining code into a tidy class and disabled unnecessary computations in the float/object case, thus getting about 10% better performance ([GH211](#))
- Improved speed of *DataFrame.xs* on mixed-type DataFrame objects by about 5x, regression from 0.3.0 ([GH215](#))
- With new *DataFrame.align* method, speeding up binary operations between differently-indexed DataFrame objects by 10-25%.
- Significantly sped up conversion of nested dict into DataFrame ([GH212](#))
- Can pass hierarchical index level name to *groupby* instead of the level number if desired ([GH223](#))
- Add support for different delimiters in *DataFrame.to\_csv* ([GH244](#))
- Add more helpful error message when importing pandas post-installation from the source directory ([GH250](#))
- Significantly speed up DataFrame *\_\_repr\_\_* and *count* on large mixed-type DataFrame objects
- Better handling of pyx file dependencies in Cython module build ([GH271](#))

### 38.39.5 Bug Fixes

- *read\_csv* / *read\_table* fixes
  - Be less aggressive about converting float->int in cases of floating point representations of integers like 1.0, 2.0, etc.
  - “True”/”False” will not get correctly converted to boolean
  - Index name attribute will get set when specifying an index column
  - Passing column names should force *header=None* ([GH257](#))
  - Don’t modify passed column names when *index\_col* is not None ([GH258](#))
  - Can sniff CSV separator in zip file (since seek is not supported, was failing before)
- Worked around matplotlib “bug” in which *series[:, np.newaxis]* fails. Should be reported upstream to matplotlib ([GH224](#))
- *DataFrame.iteritems* was not returning Series with the name attribute set. Also neither was *DataFrame.\_series*
- Can store *datetime.date* objects in *HDFStore* ([GH231](#))
- Index and Series names are now stored in *HDFStore*
- Fixed problem in which data would get upcasted to object dtype in *GroupBy.apply* operations ([GH237](#))
- Fixed outer join bug with empty DataFrame ([GH238](#))
- Can create empty Panel ([GH239](#))
- Fix join on single key when passing list with 1 entry ([GH246](#))
- Don’t raise Exception on plotting DataFrame with an all-NA column ([GH251](#), [GH254](#))
- Bug min/max errors when called on integer DataFrames ([GH241](#))
- *DataFrame.iteritems* and *DataFrame.\_series* not assigning name attribute
- *Panel.\_\_repr\_\_* raised exception on length-0 major/minor axes
- *DataFrame.join* on key with empty DataFrame produced incorrect columns
- Implemented *MultiIndex.diff* ([GH260](#))
- *Int64Index.take* and *MultiIndex.take* lost name field, fix downstream issue [GH262](#)

- Can pass list of tuples to *Series* ([GH270](#))
- Can pass level name to *DataFrame.stack*
- Support set operations between MultiIndex and Index
- Fix many corner cases in MultiIndex set operations - Fix MultiIndex-handling bug with GroupBy.apply when returned groups are not indexed the same
- Fix corner case bugs in DataFrame.apply
- Setting DataFrame index did not cause Series cache to get cleared
- Various int32 -> int64 platform-specific issues
- Don't be too aggressive converting to integer when parsing file with MultiIndex ([GH285](#))
- Fix bug when slicing Series with negative indices before beginning

### 38.39.6 Thanks

- Thomas Kluyver
- Daniel Fortunov
- Aman Thakral
- Luca Beltrame
- Wouter Overmeire

## 38.40 pandas 0.4.3

**Release date:** 10/9/2011

This is largely a bugfix release from 0.4.2 but also includes a handful of new and enhanced features. Also, pandas can now be installed and used on Python 3 (thanks Thomas Kluyver!).

### 38.40.1 New Features

- Python 3 support using 2to3 ([GH200](#), Thomas Kluyver)
- Add *name* attribute to *Series* and added relevant logic and tests. Name now prints as part of *Series.\_\_repr\_\_*
- Add *name* attribute to standard Index so that stacking / unstacking does not discard names and so that indexed DataFrame objects can be reliably round-tripped to flat files, pickle, HDF5, etc.
- Add *isnull* and *notnull* as instance methods on Series ([GH209](#), [GH203](#))

### 38.40.2 Improvements to existing features

- Skip xldr-related unit tests if not installed
- *Index.append* and *MultiIndex.append* can accept a list of Index objects to concatenate together
- Altered binary operations on differently-indexed SparseSeries objects to use the integer-based (dense) alignment logic which is faster with a larger number of blocks ([GH205](#))
- Refactored *Series.\_\_repr\_\_* to be a bit more clean and consistent

### 38.40.3 API Changes

- *Series.describe* and *DataFrame.describe* now bring the 25% and 75% quartiles instead of the 10% and 90% deciles. The other outputs have not changed
- *Series.toString* will print deprecation warning, has been de-camelCased to *to\_string*

### 38.40.4 Bug Fixes

- Fix broken interaction between *Index* and *Int64Index* when calling *intersection*. Implement *Int64Index.intersection*
- *MultiIndex.sortlevel* discarded the level names (GH202)
- Fix bugs in *groupby*, *join*, and *append* due to improper concatenation of *MultiIndex* objects (GH201)
- Fix regression from 0.4.1, *isnull* and *notnull* ceased to work on other kinds of Python scalar objects like *datetime.datetime*
- Raise more helpful exception when attempting to write empty *DataFrame* or *LongPanel* to *HDFStore* (GH204)
- Use *stdlib csv* module to properly escape strings with commas in *DataFrame.to\_csv* (GH206, Thomas Kluyver)
- Fix Python *ndarray* access in Cython code for sparse blocked index integrity check
- Fix bug writing *Series* to CSV in Python 3 (GH209)
- Miscellaneous Python 3 bugfixes

### 38.40.5 Thanks

- Thomas Kluyver
- rsamson

## 38.41 pandas 0.4.2

**Release date:** 10/3/2011

This is a performance optimization release with several bug fixes. The new *Int64Index* and new merging / joining Cython code and related Python infrastructure are the main new additions

### 38.41.1 New Features

- Added fast *Int64Index* type with specialized *join*, *union*, *intersection*. Will result in significant performance enhancements for int64-based time series (e.g. using NumPy's *datetime64* one day) and also faster operations on *DataFrame* objects storing record array-like data.
- Refactored *Index* classes to have a *join* method and associated data alignment routines throughout the codebase to be able to leverage optimized joining / merging routines.
- Added *Series.align* method for aligning two series with choice of join method
- Wrote faster Cython data alignment / merging routines resulting in substantial speed increases
- Added *is\_monotonic* property to *Index* classes with associated Cython code to evaluate the monotonicity of the *Index* values



- Add method *get\_level\_values* to *MultiIndex*
- Implemented shallow copy of *BlockManager* object in *DataFrame* internals

### 38.41.2 Improvements to existing features

- Improved performance of *isnull* and *notnull*, a regression from v0.3.0 ([GH187](#))
- Wrote templating / code generation script to auto-generate Cython code for various functions which need to be available for the 4 major data types used in pandas (float64, bool, object, int64)
- Refactored code related to *DataFrame.join* so that intermediate aligned copies of the data in each *DataFrame* argument do not need to be created. Substantial performance increases result ([GH176](#))
- Substantially improved performance of generic *Index.intersection* and *Index.union*
- Improved performance of *DateRange.union* with overlapping ranges and non-cacheable offsets (like Minute). Implemented analogous fast *DateRange.intersection* for overlapping ranges.
- Implemented *BlockManager.take* resulting in significantly faster *take* performance on mixed-type *DataFrame* objects ([GH104](#))
- Improved performance of *Series.sort\_index*
- Significant groupby performance enhancement: removed unnecessary integrity checks in *DataFrame* internals that were slowing down slicing operations to retrieve groups
- Added informative Exception when passing dict to *DataFrame* groupby aggregation with *axis != 0*

### 38.41.3 API Changes

#### 38.41.4 Bug Fixes

- Fixed minor unhandled exception in Cython code implementing fast groupby aggregation operations
- Fixed bug in unstacking code manifesting with more than 3 hierarchical levels
- Throw exception when step specified in label-based slice ([GH185](#))
- Fix *isnull* to correctly work with *np.float32*. Fix upstream bug described in [GH182](#)
- Finish implementation of *as\_index=False* in groupby for *DataFrame* aggregation ([GH181](#))
- Raise *SkipTest* for pre-epoch *HDFStore* failure. Real fix will be sorted out via *datetime64* dtype

### 38.41.5 Thanks

- Uri Laserson
- Scott Sinclair

## 38.42 pandas 0.4.1

**Release date:** 9/25/2011

This is primarily a bug fix release but includes some new features and improvements

### 38.42.1 New Features

- Added new *DataFrame* methods *get\_dtype\_counts* and property *dtypes*
- Setting of values using *.ix* indexing attribute in mixed-type *DataFrame* objects has been implemented (fixes [GH135](#))
- *read\_csv* can read multiple columns into a *MultiIndex*. *DataFrame*'s *to\_csv* method will properly write out a *MultiIndex* which can be read back ([GH151](#), thanks to Skipper Seabold)
- Wrote fast time series merging / joining methods in Cython. Will be integrated later into *DataFrame.join* and related functions
- Added *ignore\_index* option to *DataFrame.append* for combining unindexed records stored in a *DataFrame*

### 38.42.2 Improvements to existing features

- Some speed enhancements with internal Index type-checking function
- *DataFrame.rename* has a new *copy* parameter which can rename a *DataFrame* in place
- Enable unstacking by level name ([GH142](#))
- Enable sortlevel to work by level name ([GH141](#))
- *read\_csv* can automatically “sniff” other kinds of delimiters using *csv.Sniffer* ([GH146](#))
- Improved speed of unit test suite by about 40%
- Exception will not be raised calling *HDFStore.remove* on non-existent node with where clause
- Optimized *\_ensure\_index* function resulting in performance savings in type-checking Index objects

### 38.42.3 API Changes

### 38.42.4 Bug Fixes

- Fixed *DataFrame* constructor bug causing downstream problems (e.g. *.copy()* failing) when passing a *Series* as the values along with a column name and index
- Fixed single-key groupby on *DataFrame* with *as\_index=False* ([GH160](#))
- *Series.shift* was failing on integer *Series* ([GH154](#))
- *unstack* methods were producing incorrect output in the case of duplicate hierarchical labels. An exception will now be raised ([GH147](#))
- Calling *count* with level argument caused reduceat failure or segfault in earlier NumPy ([GH169](#))
- Fixed *DataFrame.corrwith* to automatically exclude non-numeric data (GH [GH144](#))
- Unicode handling bug fixes in *DataFrame.to\_string* ([GH138](#))
- Excluding OLS degenerate unit test case that was causing platform specific failure ([GH149](#))
- Skip *blosc*-dependent unit tests for *PyTables* < 2.2 ([GH137](#))
- Calling *copy* on *DateRange* did not copy over attributes to the new object ([GH168](#))
- Fix bug in *HDFStore* in which Panel data could be appended to a Table with different item order, thus resulting in an incorrect result read back

### 38.42.5 Thanks

- Yaroslav Halchenko
- Jeff Reback
- Skipper Seabold
- Dan Lovell
- Nick Pentreath

## 38.43 pandas 0.4.0

Release date: 9/12/2011

### 38.43.1 New Features

- *pandas.core.sparse* module: “Sparse” (mostly-NA, or some other fill value) versions of *Series*, *DataFrame*, and *Panel*. For low-density data, this will result in significant performance boosts, and smaller memory footprint. Added *to\_sparse* methods to *Series*, *DataFrame*, and *Panel*. See online documentation for more on these
- Fancy indexing operator on *Series* / *DataFrame*, e.g. via *.ix* operator. Both getting and setting of values is supported; however, setting values will only currently work on homogeneously-typed *DataFrame* objects. Things like:
  - `series.ix[[d1, d2, d3]]`
  - `frame.ix[5:10, ['C', 'B', 'A']], frame.ix[5:10, 'A':'C']`
  - `frame.ix[date1:date2]`
- Significantly enhanced *groupby* functionality
  - Can groupby multiple keys, e.g. `df.groupby(['key1', 'key2'])`. Iteration with multiple groupings products a flattened tuple
  - “Nuisance” columns (non-aggregatable) will automatically be excluded from *DataFrame* aggregation operations
  - Added automatic “dispatching to *Series* / *DataFrame* methods to more easily invoke methods on groups. e.g. `s.groupby(crit).std()` will work even though *std* is not implemented on the *GroupBy* class
- Hierarchical / multi-level indexing
  - New the *MultiIndex* class. Integrated *MultiIndex* into *Series* and *DataFrame* fancy indexing, slicing, `__getitem__` and `__setitem__`, reindexing, etc. Added *level* keyword argument to *groupby* to enable grouping by a level of a *MultiIndex*
- New data reshaping functions: *stack* and *unstack* on *DataFrame* and *Series*
  - Integrate with *MultiIndex* to enable sophisticated reshaping of data
- *Index* objects (labels for axes) are now capable of holding tuples
- *Series.describe*, *DataFrame.describe*: produces an R-like table of summary statistics about each data column
- *DataFrame.quantile*, *Series.quantile* for computing sample quantiles of data across requested axis
- Added general *DataFrame.dropna* method to replace *dropIncompleteRows* and *dropEmptyRows*, deprecated those.

- *Series* arithmetic methods with optional `fill_value` for missing data, e.g. `a.add(b, fill_value=0)`. If a location is missing for both it will still be missing in the result though.
- `fill_value` option has been added to *DataFrame*.{`add`, `mul`, `sub`, `div`} methods similar to *Series*
- Boolean indexing with *DataFrame* objects: `data[data > 0.1] = 0.1` or `data[data > other] = 1`.
- *pytz* / *tzinfo* support in *DateRange*
  - `tz_localize`, `tz_normalize`, and `tz_validate` methods added
- Added *ExcelFile* class to *pandas.io.parsers* for parsing multiple sheets out of a single Excel 2003 document
- *GroupBy* aggregations can now optionally *broadcast*, e.g. produce an object of the same size with the aggregated value propagated
- Added *select* function in all data structures: *reindex* axis based on arbitrary criterion (function returning boolean value), e.g. `frame.select(lambda x: 'foo' in x, axis=1)`
- *DataFrame consolidate* method, API function relating to redesigned internals
- *DataFrame.insert* method for inserting column at a specified location rather than the default `__setitem__` behavior (which puts it at the end)
- *HDFStore* class in *pandas.io.pytables* has been largely rewritten using patches from Jeff Reback from others. It now supports mixed-type *DataFrame* and *Series* data and can store *Panel* objects. It also has the option to query *DataFrame* and *Panel* data. Loading data from legacy *HDFStore* files is supported explicitly in the code
- Added *set\_printoptions* method to modify appearance of *DataFrame* tabular output
- *rolling\_quantile* functions; a moving version of *Series.quantile* / *DataFrame.quantile*
- Generic *rolling\_apply* moving window function
- New *drop* method added to *Series*, *DataFrame*, etc. which can drop a set of labels from an axis, producing a new object
- *reindex* methods now sport a *copy* option so that data is not forced to be copied then the resulting object is indexed the same
- Added *sort\_index* methods to *Series* and *Panel*. Renamed *DataFrame.sort* to *sort\_index*. Leaving *DataFrame.sort* for now.
- Added *skipna* option to statistical instance methods on all the data structures
- *pandas.io.data* module providing a consistent interface for reading time series data from several different sources

### 38.43.2 Improvements to existing features

- The 2-dimensional *DataFrame* and *DataMatrix* classes have been extensively redesigned internally into a single class *DataFrame*, preserving where possible their optimal performance characteristics. This should reduce confusion from users about which class to use.
  - Note that under the hood there is a new essentially “lazy evaluation” scheme within respect to adding columns to *DataFrame*. During some operations, like-typed blocks will be “consolidated” but not before.
- *DataFrame* accessing columns repeatedly is now significantly faster than *DataMatrix* used to be in 0.3.0 due to an internal *Series* caching mechanism (which are all views on the underlying data)
- Column ordering for mixed type data is now completely consistent in *DataFrame*. In prior releases, there was inconsistent column ordering in *DataMatrix*
- Improved console / string formatting of *DataMatrix* with negative numbers
- Improved tabular data parsing functions, *read\_table* and *read\_csv*:

- Added *skiprows* and *na\_values* arguments to *pandas.io.parsers* functions for more flexible IO
- *parseCSV* / *read\_csv* functions and others in *pandas.io.parsers* now can take a list of custom NA values, and also a list of rows to skip
- Can slice *DataFrame* and get a view of the data (when homogeneously typed), e.g. `frame.xs(idx, copy=False)` or `frame.ix[idx]`
- Many speed optimizations throughout *Series* and *DataFrame*
- Eager evaluation of groups when calling *groupby* functions, so if there is an exception with the grouping function it will be raised immediately versus sometime later on when the groups are needed
- *datetools.WeekOfMonth* offset can be parameterized with *n* different than 1 or -1.
- Statistical methods on *DataFrame* like *mean*, *std*, *var*, *skew* will now ignore non-numerical data. Before a not very useful error message was generated. A flag *numeric\_only* has been added to *DataFrame.sum* and *DataFrame.count* to enable this behavior in those methods if so desired (disabled by default)
- *DataFrame.pivot* generalized to enable pivoting multiple columns into a *DataFrame* with hierarchical columns
- *DataFrame* constructor can accept structured / record arrays
- *Panel* constructor can accept a dict of *DataFrame*-like objects. Do not need to use *from\_dict* anymore (*from\_dict* is there to stay, though).

### 38.43.3 API Changes

- The *DataMatrix* variable now refers to *DataFrame*, will be removed within two releases
- *WidePanel* is now known as *Panel*. The *WidePanel* variable in the pandas namespace now refers to the renamed *Panel* class
- *LongPanel* and *Panel* / *WidePanel* now no longer have a common subclass. *LongPanel* is now a subclass of *DataFrame* having a number of additional methods and a hierarchical index instead of the old *LongPanelIndex* object, which has been removed. Legacy *LongPanel* pickles may not load properly
- Cython is now required to build *pandas* from a development branch. This was done to avoid continuing to check in cythonized C files into source control. Builds from released source distributions will not require Cython
- Cython code has been moved up to a top level *pandas/src* directory. Cython extension modules have been renamed and promoted from the *lib* subpackage to the top level, i.e.
  - *pandas.lib.tseries* -> *pandas.\_tseries*
  - *pandas.lib.sparse* -> *pandas.\_sparse*
- *DataFrame* pickling format has changed. Backwards compatibility for legacy pickles is provided, but it's recommended to consider PyTables-based *HDFStore* for storing data with a longer expected shelf life
- A *copy* argument has been added to the *DataFrame* constructor to avoid unnecessary copying of data. Data is no longer copied by default when passed into the constructor
- Handling of boolean dtype in *DataFrame* has been improved to support storage of boolean data with NA / NaN values. Before it was being converted to float64 so this should not (in theory) cause API breakage
- To optimize performance, Index objects now only check that their labels are unique when uniqueness matters (i.e. when someone goes to perform a lookup). This is a potentially dangerous tradeoff, but will lead to much better performance in many places (like *groupby*).
- Boolean indexing using *Series* must now have the same indices (labels)
- Backwards compatibility support for *begin/end/nPeriods* keyword arguments in *DateRange* class has been removed

- More intuitive / shorter filling aliases *ffill* (for *pad*) and *bfill* (for *backfill*) have been added to the functions that use them: *reindex*, *asfreq*, *fillna*.
- *pandas.core.mixins* code moved to *pandas.core.generic*
- *buffer* keyword arguments (e.g. *DataFrame.toString*) renamed to *buf* to avoid using Python built-in name
- *DataFrame.rows()* removed (use *DataFrame.index*)
- Added deprecation warning to *DataFrame.cols()*, to be removed in next release
- *DataFrame* deprecations and de-camelCasing: *merge*, *asMatrix*, *toDataMatrix*, *\_firstTimeWithValue*, *\_lastTimeWithValue*, *toRecords*, *fromRecords*, *tgrouphy*, *toString*
- *pandas.io.parsers* method deprecations
  - *parseCSV* is now *read\_csv* and keyword arguments have been de-camelCased
  - *parseText* is now *read\_table*
  - *parseExcel* is replaced by the *ExcelFile* class and its *parse* method
- *fillMethod* arguments (deprecated in prior release) removed, should be replaced with *method*
- *Series.fill*, *DataFrame.fill*, and *Panel.fill* removed, use *fillna* instead
- *groupby* functions now exclude NA / NaN values from the list of groups. This matches R behavior with NAs in factors e.g. with the *tapply* function
- Removed *parseText*, *parseCSV* and *parseExcel* from pandas namespace
- *Series.combineFunc* renamed to *Series.combine* and made a bit more general with a *fill\_value* keyword argument defaulting to NaN
- Removed *pandas.core.pytools* module. Code has been moved to *pandas.core.common*
- Tacked on *groupName* attribute for groups in *GroupBy* renamed to *name*
- *Panel/LongPanel dims* attribute renamed to *shape* to be more conformant
- Slicing a *Series* returns a view now
- More Series deprecations / renaming: *toCSV* to *to\_csv*, *asOf* to *asof*, *merge* to *map*, *applymap* to *apply*, *toDict* to *to\_dict*, *combineFirst* to *combine\_first*. Will print *FutureWarning*.
- *DataFrame.to\_csv* does not write an “index” column label by default anymore since the output file can be read back without it. However, there is a new *index\_label* argument. So you can do *index\_label='index'* to emulate the old behavior
- *datetools.Week* argument renamed from *dayOfWeek* to *weekday*
- *timeRule* argument in *shift* has been deprecated in favor of using the *offset* argument for everything. So you can still pass a time rule string to *offset*
- Added optional *encoding* argument to *read\_csv*, *read\_table*, *to\_csv*, *from\_csv* to handle unicode in Python 2.x

### 38.43.4 Bug Fixes

- Column ordering in *pandas.io.parsers.parseCSV* will match CSV in the presence of mixed-type data
- Fixed handling of Excel 2003 dates in *pandas.io.parsers*
- *DateRange* caching was happening with high resolution *DateOffset* objects, e.g. *DateOffset(seconds=1)*. This has been fixed
- Fixed *\_\_truediv\_\_* issue in *DataFrame*

- Fixed *DataFrame.toCSV* bug preventing IO round trips in some cases
- Fixed bug in *Series.plot* causing matplotlib to barf in exceptional cases
- Disabled *Index* objects from being hashable, like ndarrays
- Added `__ne__` implementation to *Index* so that operations like `ts[ts != idx]` will work
- Added `__ne__` implementation to *DataFrame*
- Bug / unintuitive result when calling *fillna* on unordered labels
- Bug calling *sum* on boolean *DataFrame*
- Bug fix when creating a *DataFrame* from a dict with scalar values
- `Series.{sum, mean, std, ...}` now return NA/NaN when the whole Series is NA
- NumPy 1.4 through 1.6 compatibility fixes
- Fixed bug in bias correction in *rolling\_cov*, was affecting *rolling\_corr* too
- R-square value was incorrect in the presence of fixed and time effects in the *PanelOLS* classes
- *HDFStore* can handle duplicates in table format, will take

### 38.43.5 Thanks

- Joon Ro
- Michael Pennington
- Chris Uga
- Chris Withers
- Jeff Reback
- Ted Square
- Craig Austin
- William Ferreira
- Daniel Fortunov
- Tony Roberts
- Martin Felder
- John Marino
- Tim McNamara
- Justin Berka
- Dieter Vandenbussche
- Shane Conway
- Skipper Seabold
- Chris Jordan-Squire

## 38.44 pandas 0.3.0

Release date: February 20, 2011

### 38.44.1 New features

- *corrwith* function to compute column- or row-wise correlations between two DataFrame objects
- Can boolean-index DataFrame objects, e.g. `df[df > 2] = 2`, `px[px > last_px] = 0`
- Added comparison magic methods (`__lt__`, `__gt__`, etc.)
- Flexible explicit arithmetic methods (`add`, `mul`, `sub`, `div`, etc.)
- Added *reindex\_like* method
- Added *reindex\_like* method to WidePanel
- Convenience functions for accessing SQL-like databases in *pandas.io.sql* module
- Added (still experimental) HDFStore class for storing pandas data structures using HDF5 / PyTables in *pandas.io.pytables* module
- Added WeekOfMonth date offset
- *pandas.rpy* (experimental) module created, provide some interfacing / conversion between rpy2 and pandas

### 38.44.2 Improvements to existing features

- Unit test coverage: 100% line coverage of core data structures
- Speed enhancement to `rolling_{median, max, min}`
- Column ordering between DataFrame and DataMatrix is now consistent: before DataFrame would not respect column order
- Improved `{Series, DataFrame}.plot` methods to be more flexible (can pass matplotlib Axis arguments, plot DataFrame columns in multiple subplots, etc.)

### 38.44.3 API Changes

- Exponentially-weighted moment functions in *pandas.stats.moments* have a more consistent API and accept a `min_periods` argument like their regular moving counterparts.
- `fillMethod` argument in Series, DataFrame changed to `method`, *FutureWarning* added.
- `fill` method in Series, DataFrame/DataMatrix, WidePanel renamed to `fillna`, *FutureWarning* added to `fill`
- Renamed `DataFrame.getXS` to `xs`, *FutureWarning* added
- Removed `cap` and `floor` functions from DataFrame, renamed to `clip_upper` and `clip_lower` for consistency with NumPy



### 38.44.4 Bug Fixes

- Fixed bug in `IndexableSkiplist` Cython code that was breaking `rolling_max` function
- Numerous `numpy.int64`-related indexing fixes
- Several NumPy 1.4.0 NaN-handling fixes
- Bug fixes to `pandas.io.parsers.parseCSV`
- Fixed `DateRange` caching issue with unusual date offsets
- Fixed bug in `DateRange.union`
- Fixed corner case in `IndexableSkiplist` implementation



## BIBLIOGRAPHY

- [R16] <http://docs.sqlalchemy.org>
- [R17] <https://www.python.org/dev/peps/pep-0249/>
- [R27] <https://docs.python.org/3/library/pickle.html>
- [R28] <http://docs.sqlalchemy.org>
- [R29] <https://www.python.org/dev/peps/pep-0249/>
- [R15] <https://docs.python.org/3/library/pickle.html>
- [R21] <https://docs.python.org/3/library/pickle.html>
- [R22] <http://docs.sqlalchemy.org>
- [R23] <https://www.python.org/dev/peps/pep-0249/>
- [R30] [https://en.wikipedia.org/wiki/Imputation\\_\(statistics\)](https://en.wikipedia.org/wiki/Imputation_(statistics))
- [R31] [https://en.wikipedia.org/wiki/Imputation\\_\(statistics\)](https://en.wikipedia.org/wiki/Imputation_(statistics))
- [R32] [https://en.wikipedia.org/wiki/Imputation\\_\(statistics\)](https://en.wikipedia.org/wiki/Imputation_(statistics))
- [R33] “Bootstrapping (statistics)” in [https://en.wikipedia.org/wiki/Bootstrapping\\_\(statistics\)](https://en.wikipedia.org/wiki/Bootstrapping_(statistics))



## PYTHON MODULE INDEX

### p

pandas, [1](#)