pandas: powerful Python data analysis toolkit

Release 0.11.0.dev-9988e5f

Wes McKinney & PyData Development Team

April 23, 2013
6 Essential basic functionality
  6.1 Head and Tail ................................................. 87
  6.2 Attributes and the raw ndarray(s) ............................... 88
  6.3 Flexible binary operations ..................................... 89
  6.4 Descriptive statistics ......................................... 93
  6.5 Function application .......................................... 99
  6.6 Reindexing and altering labels ................................. 102
  6.7 Iteration .................................................. 108
  6.8 Vectorized string methods ................................... 110
  6.9 Sorting by index and value ................................... 113
  6.10 Copying, type casting ......................................... 115
  6.11 Pickling and serialization .................................... 116
  6.12 Working with package options ................................. 117
  6.13 Console Output Formatting .................................... 120

7 Indexing and selecting data
  7.1 Basics .................................................. 121
  7.2 Advanced indexing with labels ................................. 132
  7.3 Index objects ............................................... 136
  7.4 Hierarchical indexing (MultiIndex) ............................. 137
  7.5 Adding an index to an existing DataFrame ...................... 148
  7.6 Indexing internal details ..................................... 150

8 Computational tools
  8.1 Statistical functions .......................................... 153
  8.2 Moving (rolling) statistics / moments ......................... 157
  8.3 Expanding window moment functions ............................ 163
  8.4 Exponentially weighted moment functions ...................... 165
  8.5 Linear and panel regression .................................. 166

9 Working with missing data
  9.1 Missing data basics .......................................... 173
  9.2 Calculations with missing data ................................ 175
  9.3 Cleaning / filling missing data ................................ 176
  9.4 Missing data casting rules and indexing ....................... 182

10 Group By: split-apply-combine
  10.1 Splitting an object into groups ............................... 185
  10.2 Iterating through groups ..................................... 189
  10.3 Aggregation ............................................... 190
  10.4 Transformation ............................................ 193
  10.5 Dispatching to instance methods ............................... 196
  10.6 Flexible apply ............................................. 197
  10.7 Other useful features ....................................... 199

11 Merge, join, and concatenate
  11.1 Concatenating objects ....................................... 201
  11.2 Database-style DataFrame joining/merging ...................... 210

12 Reshaping and Pivot Tables
  12.1 Reshaping by pivoting DataFrame objects ...................... 219
  12.2 Reshaping by stacking and unstacking ......................... 220
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 v0.10.1 (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.

1.1.1 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)

1.1.2 New features

- MySQL support for database (contribution from Dan Allan)

1.1.3 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 [1530]: store = HDFStore('store.h5')

In [1531]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
                           columns=['A', 'B', 'C'])

In [1532]: df['string'] = 'foo'

In [1533]: df.ix[4:6,'string'] = np.nan

In [1534]: df.ix[7:9,'string'] = 'bar'

In [1535]: df['string2'] = 'cool'

In [1536]: df
Out[1536]:
      A         B          C    string string2
2000-01-01  0.741687  0.035967 -2.700230     foo    cool
2000-01-02  0.777316  1.201654  0.775594     foo    cool
2000-01-03  0.916695 -0.511978  0.805595     foo    cool
2000-01-04 -0.517789 -0.980332 -1.325032     foo    cool
2000-01-05  0.015397  1.063654 -0.297355    NaN    cool
2000-01-06  1.118334 -1.750153  0.507924    NaN    cool
2000-01-07 -0.163195  0.285564 -0.332279     foo    cool
2000-01-08 -0.516040 -0.531297 -0.409554    NaN    cool

# on-disk operations
In [1537]: store.append('df', df, data_columns = ['B','C','string','string2'])

In [1538]: store.select('df', ['B > 0', 'string == foo'])
Out[1538]:
      A         B          C    string string2
2000-01-01  0.741687  0.035967 -2.700230     foo    cool
2000-01-02  0.777316  1.201654  0.775594     foo    cool
2000-01-07 -0.163195  0.285564 -0.332279     foo    cool

# this is in-memory version of this type of selection
In [1539]: df[(df.B > 0) & (df.string == 'foo')]
Out[1539]:
      A         B          C    string string2
2000-01-01  0.741687  0.035967 -2.700230     foo    cool
2000-01-02  0.777316  1.201654  0.775594     foo    cool
2000-01-07 -0.163195  0.285564 -0.332279     foo    cool

Retrieving unique values in an indexable or data column.

In [1540]: store.unique('df','index')
Out[1540]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-08 00:00:00]
Length: 8, Freq: None, Timezone: None

In [1541]: store.unique('df','string')
Out[1541]: Index([bar, foo], dtype=object)

You can now store datetime64 in data columns

In [1542]: df_mixed = df.copy()
In [1543]: df_mixed['datetime64'] = Timestamp('20010102')

In [1544]: df_mixed.ix[3:4, ['A', 'B']] = np.nan

In [1545]: store.append('df_mixed', df_mixed)

In [1546]: df_mixed1 = store.select('df_mixed')

In [1547]: df_mixed1
Out[1547]:
   A  B  C     string     string2 datetime64
0 0.741687 0.035967 -2.700230   foo  cool 2001-01-02 00:00:00
1 0.777316 1.201654  0.775594   foo  cool 2001-01-02 00:00:00
2 0.916695 -0.511978  0.805595   foo  cool 2001-01-02 00:00:00
3          NaN          NaN          NaN          NaN          NaN          NaN
4 0.015397 1.063654 -1.325032   foo  cool 2001-01-02 00:00:00
5 1.118334 -1.750153  0.507924   foo  cool 2001-01-02 00:00:00
6 -0.163195 0.285564 -0.332279   foo  cool 2001-01-02 00:00:00
7 -0.516040 -0.531297 -0.409554   bar  cool 2001-01-02 00:00:00

In [1548]: df_mixed1.get_dtype_counts()
Out[1548]:
   datetime64[ns] 1
   float64        3
   object         2
   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 [1549]: store.select('df', columns=['A', 'B'])
Out[1549]:
     A  B
   2000-01-01 0.741687 0.035967
   2000-01-02 0.777316 1.201654
   2000-01-03 0.916695 -0.511978
   2000-01-04          NaN          NaN
   2000-01-05 0.015397 1.063654
   2000-01-06 1.118334 -1.750153
   2000-01-07 -0.163195 0.285564
   2000-01-08 -0.516040 -0.531297

HDFStore now serializes multi-index dataframes when appending tables.

In [1550]:
   ...: 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 [1551]: df = DataFrame(np.random.randn(10, 3), index=index,
   ...:                   columns=['A', 'B', 'C'])
   ....:

In [1552]: df
Out[1552]:
     A  B  C
foo  bar

1.1. v0.10.1 (January 22, 2013)
foo one  0.055458 -0.000871 -0.156757  
two  -1.193604  0.768787 -0.228047  
three  0.054979 -0.423256  0.175289  
bar one  -0.961203 -0.302857  0.047525  
two  -0.987381 -0.082381  1.122844  
three  0.357760 -1.287685 -0.555503  
baz two  -1.721204 -0.040879 -1.742960  
three  -0.994435 -1.857899 -1.409501  
qux one  -1.263551 -0.952076  1.253998  
two  -0.994435 -1.857899 -1.409501  
three  2.056446  0.686683  0.295824  

In [1553]: store.append('mi', df)

In [1554]: store.select('mi')

Out[1554]:  
       A   B    C
foo bar
       A   B    C
foo one  0.055458 -0.000871 -0.156757  
two  -1.193604  0.768787 -0.228047  
three  0.054979 -0.423256  0.175289  
bar one  -0.961203 -0.302857  0.047525  
two  -0.987381 -0.082381  1.122844  
three  0.357760 -1.287685 -0.555503  
baz two  -1.721204 -0.040879 -1.742960  
three  -0.994435 -1.857899 -1.409501  
qux one  -1.263551 -0.952076  1.253998  
two  -0.994435 -1.857899 -1.409501  
three  2.056446  0.686683  0.295824

# the levels are automatically included as data columns
In [1555]: store.select('mi', Term('foo=bar'))

Out[1555]:  
       A   B    C
foo bar
       A   B    C
foo one  0.055458 -0.000871 -0.156757  
two  -1.193604  0.768787 -0.228047  
three  0.054979 -0.423256  0.175289  
bar one  -0.961203 -0.302857  0.047525  
two  -0.987381 -0.082381  1.122844  
three  0.357760 -1.287685 -0.555503  
baz two  -1.721204 -0.040879 -1.742960  
three  -0.994435 -1.857899 -1.409501  
qux one  -1.263551 -0.952076  1.253998  
two  -0.994435 -1.857899 -1.409501  
three  2.056446  0.686683  0.295824

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 [1556]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
   ...: columns=['A', 'B', 'C', 'D', 'E', 'F'])
   ...:
   In [1557]: df_mt['foo'] = 'bar'

# you can also create the tables individually
In [1558]: store.append_to_multiple({ 'df1_mt' : ['A','B'], 'df2_mt' : None }, df_mt, selector = 'df1_mt')

In [1559]: store
Out[1559]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df    frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,string,string2])
/df1_mt frame_table (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_mixed frame_table (typ->appendable,nrows->8,ncols->6,indexers->[index])
/mi     frame_table (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->[bar,...]
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

# individual tables were created
In [1560]: store.select('df1_mt')
Out[1560]:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.452273</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.388093</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.727640</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>1.973282</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.436261</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.068377</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.203168</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-0.414547</td>
</tr>
</tbody>
</table>

In [1561]: store.select('df2_mt')
Out[1561]:

|    |    |    |    | foo |
|----|----|----|----|
| 2000-01-01 | 0.585509 | 0.483793 | 1.387714 | -0.261908 | bar |
| 2000-01-02 | 0.269055 | 0.011450 | -0.104465 | -0.406944 | bar |
| 2000-01-03 | 0.478604 | 0.463990 | 1.237388 | 0.628084 | bar |
| 2000-01-04 | 0.963953 | 0.053805 | 1.182483 | 0.566182 | bar |
| 2000-01-05 | -0.320155 | 2.545145 | 0.301306 | 1.967739 | bar |
| 2000-01-06 | -1.038566 | -0.911641 | -1.172296 | 1.539279 | bar |
| 2000-01-07 | -0.836731 | 0.283662 | -0.357312 | 1.295667 | bar |
| 2000-01-08 | -0.601194 | -0.134764 | 0.280262 | -0.627031 | bar |

# as a multiple
In [1562]: store.select_as_multiple(['df1_mt','df2_mt'], where=[‘A>0’,’B>0’], selector = ‘df1_mt')
Out[1562]:

|    |    |    |    |    | foo |
|----|----|----|----|----|
| 2000-01-01 | 0.452273 | 0.853944 | 0.585509 | 0.483793 | 1.387714 | -0.261908 | bar |
| 2000-01-07 | 1.203168 | 0.564612 | -0.836731 | 0.283662 | -0.357312 | 1.295667 | bar |

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 automagically create indicies 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 expectedrows that PyTables will expected. 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.

1.2 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.

1.2.1 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
1.2.2 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 [1563]: import pandas as pd

In [1564]: df = pd.DataFrame(np.random.randn(6, 4),
                        index=pd.date_range('1/1/2000', periods=6))

In [1565]: df
Out[1565]:
   0   1   2   3
2000-01-01 1.197755 0.443238 -0.793423 0.450845
2000-01-02 -0.833944 1.497871 -0.062647 0.156242
2000-01-03  0.752988 1.193476 -1.622707 0.924629
2000-01-04  0.865121 -0.192174 -0.924645 1.035467
2000-01-05 -0.237298 -0.193078 -0.113703 -1.510585
2000-01-06  0.426243 -0.863411  0.386999  1.318817

# deprecated now
In [1566]: df - df[0]
Out[1566]:
   0   1   2   3
2000-01-01 0 -0.754517 -1.991178 -0.746911
2000-01-02 0  2.331815  0.771297  0.990186
2000-01-03 0  0.440488 -2.375695  0.171640
2000-01-04 0 -1.057295 -1.789767  0.170346
2000-01-05 0  0.044219  0.123595 -1.273288
2000-01-06 0 -1.289654 -0.039243  0.892574

# Change your code to
In [1567]: df.sub(df[0], axis=0)  # align on axis 0 (rows)
Out[1567]:
   0   1   2   3
2000-01-01 0 -0.754517 -1.991178 -0.746911
2000-01-02 0  2.331815  0.771297  0.990186
2000-01-03 0  0.440488 -2.375695  0.171640
2000-01-04 0 -1.057295 -1.789767  0.170346
2000-01-05 0  0.044219  0.123595 -1.273288
2000-01-06 0 -1.289654 -0.039243  0.892574

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).

Note:
In [1568]: dates = pd.date_range('1/1/2000', '1/5/2000', freq='4h')

In [1569]: series = Series(np.arange(len(dates)), index=dates)

In [1570]: series
Out[1570]:
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 [1571]: series.resample('D', how='sum')
Out[1571]:
2000-01-01    15
2000-01-02    51
2000-01-03    87
2000-01-04   123
2000-01-05    24
Freq: D, dtype: float64

# old behavior
In [1572]: series.resample('D', how='sum', closed='right', label='right')
Out[1572]:
2000-01-01     0
2000-01-02     21
2000-01-03     57
2000-01-04     93
2000-01-05   129
Freq: D, dtype: float64

• Infinity and negative infinity are no longer treated as NA by isnull and notnull. That they every were was a relic of early pandas. This behavior can be re-enabled globally by the mode.use_inf_as_null option:

In [1573]: s = pd.Series([1.5, np.inf, 3.4, -np.inf])

In [1574]: pd.isnull(s)
Out[1574]:
0    False
1    False
2    False
3    False
dtype: bool

In[1575]: s.fillna(0)
Out[1575]:
0    1.500000
1         inf
2    3.400000
3         -inf
dtype: float64

In[1576]: pd.set_option('use_inf_as_null', True)

In[1577]: pd.isnull(s)
Out[1577]:
0    False
1     True
2    False
3     True
dtype: bool

In[1578]: s.fillna(0)
Out[1578]:
0    1.5
1     0.0
2    3.4
3     0.0
dtype: float64

In[1579]: 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 unnecessary.

• 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[1580]:

In[1581]: print data
a,b,c
1,Yes,2
3,No,4

In[1582]: pd.read_csv(StringIO(data), header=None)
Out[1582]:
   0  1  2
0  a  b  c
1  1  Yes  2
In[1583]: pd.read_csv(StringIO(data), header=None, prefix='X')
Out[1583]:
 X0 X1 X2
 0 a b c
 1 Yes 2
 2 No 4

- 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[1584]: print data
a,b,c
1,Yes,2
3,No,4

In[1585]: pd.read_csv(StringIO(data))
Out[1585]:
a b c
0 1 Yes 2
1 3 No 4

In[1586]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[1586]:
a b c
0 1 True 2
1 3 False 4

- 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[1587]: s = Series([np.nan, 1., 2., np.nan, 4])

In[1588]: s
Out[1588]:
0   NaN
1    1
2    2
3   NaN
4    4
dtype: float64

In[1589]: s.fillna(0)
Out[1589]:
0   0
1    1
2    2
3   0
4    4
dtype: float64

In[1590]: s.fillna(method='pad')
Out[1590]:
0   NaN
1    1

Chapter 1. What's New
2
3
4
dtype: float64

Convenience methods `ffill` and `bfill` have been added:

```python
In [1591]: s.ffill()
Out[1591]
0   NaN
1     1
2     2
3     2
4     4
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

```python
In [1592]: def f(x):
       ....:     return Series([x, x**2], index=['x', 'x^2'])
       ....:

In [1593]: s = Series(np.random.rand(5))
```

```python
In [1594]: s
Out[1594]
0  0.713026
1  0.539601
2  0.046682
3  0.536308
4  0.373040
dtype: float64
```

```python
In [1595]: s.apply(f)
Out[1595]
x       x^2
0  0.713026  0.508406
1  0.539601  0.291170
2  0.046682  0.002179
3  0.536308  0.287626
4  0.373040  0.139159
```

- 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:

```python
In [1596]: get_option("display.max_rows")
Out[1596]: 100
```

- `to_string()` methods now always return unicode strings (GH2224).
# 1.2.3 New features

## 1.2.4 Wide DataFrame Printing

Instead of printing the summary information, pandas now splits the string representation across multiple rows by default:

```python
In [1597]: wide_frame = DataFrame(randn(5, 16))

In [1598]: wide_frame
Out[1598]:
   0  1  2  3  4   5  6  7  
0  0.091848 -0.318810 0.950676 -1.016290 -0.267508 0.115960 -0.615949 -0.373060
1 -0.234083 -0.254881 -0.142302 1.291962 0.876700 1.704647 0.046376 0.158167
2  0.191589 -0.243287 1.684079 -0.637764 -0.323699 -1.378458 -0.868599 1.916736
3  1.247851  0.246737 1.454094 -1.166264 -0.560671 1.027488 0.252915 -0.154549
4 -0.417236  1.721160 -0.058702 -1.297764 -0.560671 1.027488 0.252915 -0.154549
   8   9  10  11  12  13  14  15
0  0.276398 -1.947432 -1.183044 -3.030491 -1.055515 -0.177967 1.269136 0.668999
1  1.503229 -0.335678  0.157359  0.828373  0.860863  0.618679 -0.507624 -1.174443
2  1.562215  0.133222  0.345906 -1.778234 -1.223208 -0.480258 -0.285245  0.775414
3  0.181686 -0.268458 -0.124345  0.443256 -0.778424  2.147255 -0.731309  0.281577
4  0.423204 -0.006209  0.314186  0.363193  0.196151 -1.598514 -0.843566 -0.353828
```

The old behavior of printing out summary information can be achieved via the `expand_frame_repr` print option:

```python
In [1599]: pd.set_option('expand_frame_repr', False)

In [1600]: wide_frame
Out[1600]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5 entries, 0 to 4
Data columns:
0  5 non-null values
1  5 non-null values
2  5 non-null values
3  5 non-null values
4  5 non-null values
5  5 non-null values
6  5 non-null values
7  5 non-null values
8  5 non-null values
9  5 non-null values
10 5 non-null values
11 5 non-null values
12 5 non-null values
13 5 non-null values
14 5 non-null values
15 5 non-null values
dtypes: float64(16)
```

The width of each line can be changed via `line_width` (80 by default):

```python
In [1601]: pd.set_option('line_width', 40)

In [1602]: wide_frame
Out[1602]:
   0  1  2  3  
0  0.091848 -0.318810 0.950676 -1.016290
```
1 -0.234083 -0.254881 -0.142302 1.291962
2 0.191589 -0.243287 1.684079 -0.637764
3 1.247851 0.246737 1.454094 -1.166264
4 -0.417236 1.721160 -0.058702 -1.297767
5 6 7
8 9 10 11
0 -0.267508 0.115960 -0.615949 -0.373060
1 0.876700 1.704647 0.046376 0.158167
2 -0.323699 -1.378458 -0.868599 1.916736
3 -0.560671 1.027488 0.252915 -0.154549
4 0.871349 -0.177241 0.207366 2.592691
5 6 7
8 9 10 11
0 0.276398 -1.947432 -1.183044 -3.030491
1 1.503229 -0.335678 0.157359 0.828373
2 1.562215 0.133322 0.345906 -1.78234
3 0.181686 -0.268458 -0.124345 0.443256
4 0.423204 -0.006209 0.314186 0.363193
5 6 7
12 13 14 15
0 -1.055515 -0.177967 1.269136 0.668999
1 0.860863 0.618679 -0.507624 -1.174443
2 -1.223208 -0.480258 -0.285245 0.775414
3 -0.778424 2.147255 -0.731309 0.281577
4 0.196151 -1.598514 -0.843566 -0.353828

1.2.5 Updated PyTables Support

*Docs* for PyTables Table format & several enhancements to the api. Here is a taste of what to expect.

In [1603]: store = HDFStore('store.h5')

In [1604]: df = DataFrame(randn(8, 3), index=date_range('1/1/2000', periods=8),
.......:     columns=['A', 'B', 'C'])
.......:

In [1605]: df
Out[1605]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.516740</td>
<td>-2.335539</td>
<td>-0.715006</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.399224</td>
<td>0.798589</td>
<td>2.101702</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.190649</td>
<td>0.595370</td>
<td>-1.672567</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.786765</td>
<td>0.133175</td>
<td>-1.077265</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.861068</td>
<td>1.982854</td>
<td>-1.059177</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>2.050701</td>
<td>-0.615165</td>
<td>-0.601019</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-1.062777</td>
<td>-1.577586</td>
<td>-0.585584</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>1.833699</td>
<td>-0.483165</td>
<td>0.652315</td>
</tr>
</tbody>
</table>

# appending data frames
In [1606]: df1 = df[0:4]

In [1607]: df2 = df[4:]

In [1608]: store.append('df', df1)

In [1609]: store.append('df', df2)

In [1610]: store
Out[1610]:
<class 'pandas.io.pytables.HDFStore'>

1.2. v0.10.0 (December 17, 2012)
# selecting the entire store
In [1611]: store.select('df')
Out[1611]:
   A     B     C
2000-01-01  0.516740 -2.335539 -0.715006
2000-01-02  0.399224  0.798589  2.101702
2000-01-03 -0.190649  0.595370 -1.077265
2000-01-04  0.786765  0.133175 -1.077265
2000-01-05  0.861068  1.982854 -1.059177
2000-01-06  2.050701 -0.615165 -0.601019
2000-01-07 -1.062777 -1.577586 -0.585584
2000-01-08  1.833699 -0.483165  0.652315

In [1612]: 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 [1613]: wp
Out[1613]:
<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

# storing a panel
In [1614]: store.append('wp',wp)

# selecting via A QUERY
In [1615]: store.select('wp',
   ....:   [ Term('major_axis>20000102'), Term('minor_axis', '=', ['A', 'B']) ]
   ....: )
Out[1615]:
<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

# removing data from tables
In [1616]: store.remove('wp', ['major_axis', '>', wp.major_axis[3]])
Out[1616]: 4

In [1617]: store.select('wp')
Out[1617]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 4 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-04 00:00:00
Minor_axis axis: A to D

# deleting a store
In [1618]: del store['df']
In [1619]: store
Out[1619]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5

/wp          wide_table (typ->appendable,nrows->16,ncols->2,indexers->[major_axis,minor_axis])

Enhancements

• added ability to hierarchical keys

In [1620]: store.put('foo/bar/bah', df)
In [1621]: store.append('food/orange', df)
In [1622]: store.append('food/apple', df)

In [1623]: store
Out[1623]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp          wide_table (typ->appendable,nrows->16,ncols->2,indexers->[major_axis,minor_axis])
/food/apple  frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah  frame (shape->[8,3])

# remove all nodes under this level
In [1624]: store.remove('food')

In [1625]: store
Out[1625]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/wp          wide_table (typ->appendable,nrows->16,ncols->2,indexers->[major_axis,minor_axis])
/foo/bar/bah  frame (shape->[8,3])

• added mixed-dtype support!

In [1626]: df['string'] = 'string'
In [1627]: df['int'] = 1
In [1628]: store.append('df',df)
In [1629]: df1 = store.select('df')

In [1630]: df1
Out[1630]:
   A    B     C  string  int
0 2000-01-01  0.516740 -2.335539  string 1
1 2000-01-02 -0.399224  0.798589  string 1
2 2000-01-03 -0.190649  0.595370  string 1
3 2000-01-04  0.786765 -0.615165  string 1
4 2000-01-05  0.610668  1.982854  string 1
5 2000-01-06  2.050701 -0.615165  string 1
6 2000-01-07 -1.062777 -1.577586  string 1
7 2000-01-08  1.833699 -0.483165  string 1

In [1631]: df1.get_dtype_counts()
float64 3
int64 1
object 1
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.

1.2.6 N Dimensional Panels (Experimental)

Adding experimental support for Panel4D and factory functions to create n-dimensional named panels. Docs for NDim. Here is a taste of what to expect.

```python
In [1632]: pd4 = 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 [1633]: pd4
Out[1633]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: Labell to Label2
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
```
See the full release notes or issue tracker on GitHub for a complete list.

1.3 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.

1.3.1 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 [1634]: df = DataFrame(np.random.randint(0, 2, (6, 3)), columns=['A', 'B', 'C'])

  In [1635]: df.sort(['A', 'B'], ascending=[1, 0])

  Out[1635]:
  A  B  C
  0  1  0
  2  1  1
  3  1  1
  4  1  1
  0  1  0
  5  1  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 [1636]: df = DataFrame(np.random.randn(6, 3), columns=['A', 'B', 'C'])

  In [1637]: df.ix[2:4] = np.nan

  In [1638]: df.rank()

  Out[1638]:
  A  B  C
  0  1  3  2
  1  2  1  1
  2 NaN NaN NaN
  3 NaN NaN NaN
  4 NaN NaN NaN
  5  3  2  3

  In [1639]: df.rank(na_option='top')

  Out[1639]:
  A  B  C
  0  4  6  5
  1  5  4  4
  2  2  2  2
  3  2  2  2
  4  2  2  2
  5  6  5  6

  In [1640]: df.rank(na_option='bottom')

  Out[1640]:
  A  B  C
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 [1641]: df = DataFrame(np.random.randn(5, 3), columns=['A','B','C'])

In [1642]: df
Out[1642]:
   A         B         C
0 -1.381185  0.365239 -1.810632
1 -0.673382 -1.967580 -0.401183
2 -0.583047 -0.998625 -0.629277
3 -0.548001 -0.852612 -0.126250
4  1.765997 -1.593297 -0.966162

In [1643]: df[df['A'] > 0]
Out[1643]:
   A         B         C
4  1.765997 -1.593297 -0.966162

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 [1644]: df[df>0]
Out[1644]:
   A         B         C
0 NaN       0.365239 NaN
1 NaN       NaN       NaN
2 NaN       NaN       NaN
3 NaN       NaN       NaN
4 1.765997  NaN       NaN

In [1645]: df.where(df>0)
Out[1645]:
   A         B         C
0 NaN       0.365239 NaN
1 NaN       NaN       NaN
2 NaN       NaN       NaN
3 NaN       NaN       NaN
4 1.765997  NaN       NaN

In [1646]: df.where(df>0,-df)
Out[1646]:
   A         B         C
0 1.381185  0.365239  1.810632
1 0.673382  1.967580  0.401183
2 0.583047  0.998625  0.629277
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 [1647]: df2 = df.copy()
In [1648]: df2[ (df2[1:4] > 0 ) ] = 3
In [1649]: df2
Out[1649]:
   A          B          C
0 -1.381185  0.365239  -1.810632
1 -0.673382 -1.967580  -0.401183
2 -0.583047 -0.998625  -0.629277
3 -0.548001 -0.852612  -0.126250
4  1.765997 -1.593297  -0.966162
```

*DataFrame.mask* is the inverse boolean operation of *where*.

```
In [1650]: df.mask(df<=0)
Out[1650]:
   A          B          C
0  NaN      0.365239     NaN
1  NaN          NaN     NaN
2  NaN          NaN     NaN
3  NaN          NaN     NaN
4  1.765997     NaN     NaN
```

- Enable referencing of Excel columns by their column names (GH1936)

  ```
  In [1651]: xl = ExcelFile(‘data/test.xls’)
  In [1652]: xl.parse(‘Sheet1’, index_col=0, parse_dates=True,
   ........: parse_cols=’A:D’
   ........:
  Out[1652]:
   A          B          C
  2000-01-03  0.980269  3.685731  -0.364217
  2000-01-04  1.047916 -0.041232  -0.161812
  2000-01-05  0.498581  0.731168  -0.537677
  2000-01-06  1.120202  1.567621   0.003641
  2000-01-07 -0.487094  0.571455  -1.611639
  2000-01-10  0.836649  0.246462   0.588543
  2000-01-11 -0.157161  1.340307   1.195778
  ```

- 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)
1.3.2 API changes

- Upsampling data with a PeriodIndex will result in a higher frequency TimeSeries that spans the original time window.

  ```python
  In [1653]: prng = period_range('2012Q1', periods=2, freq='Q')
  In [1654]: s = Series(np.random.randn(len(prng)), prng)
  In [1655]: s.resample('M')
  Out[1655]:
  2012-01 -0.332601
  2012-02 NaN
  2012-03 NaN
  2012-04 -1.327330
  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)

  ```python
  In [1656]: p = Period('2012')
  In [1657]: p.end_time
  Out[1657]: <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)

  ```python
  In [1658]: data = 'A,B,C
  00001,001,5
  00002,002,6'
  In [1659]: from cStringIO import StringIO
  In [1660]: read_csv(StringIO(data), converters={'A': lambda x: x.strip()})
  Out[1660]:
  A  B  C
  0  00001  1  5
  1  00002  2  6
  ```

See the full release notes or issue tracker on GitHub for a complete list.

1.4 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.

1.4.1 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)

1.4.2 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:

```python
In [1661]: from StringIO import StringIO
In [1662]: data = '0,0,1
1,1,0
0,1,0'
In [1663]: df = read_csv(StringIO(data), header=None)
```

```python
In [1664]: df
Out[1664]:
0 1 2
0 0 0 1
1 1 1 0
2 0 1 0
```

• 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:

```python
In [1665]: s1 = Series([1, 2, 3])
In [1666]: s1
Out[1666]:
0 1
1 2
2 3
dtype: int64
In [1667]: s2 = Series(s1, index=['foo', 'bar', 'baz'])
```

```python
In [1668]: s2
Out[1668]:
foo NaN
bar NaN
baz NaN
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)

1.4. v0.9.0 (October 7, 2012)
• Empty DataFrame columns are now created as object dtype. This will prevent a class of TypeErrors 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.

1.5 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.

1.5.1 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)

1.5.2 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
1.6 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.

1.6.1 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)

1.6.2 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.

1.6.3 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 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

1.6.4 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 :ref:`dayfirst <io.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

1.6.5 New plotting methods

Series.plot now supports a secondary_y option:

In [1669]: plt.figure()
Out[1669]: <matplotlib.figure.Figure at 0x1de229d0>

In [1670]: fx['FR'].plot(style='g')
Out[1670]: <matplotlib.axes.AxesSubplot at 0x1de22f50>

In [1671]: fx['IT'].plot(style='k--', secondary_y=True)
Out[1671]: <matplotlib.axes.AxesSubplot at 0x1de22f50>
Vytautas Jancauskas, the 2012 GSOC participant, has added many new plot types. For example, ‘kde’ is a new option:

```python
In [1672]: s = Series(np.concatenate((np.random.randn(1000),
          ....: np.random.randn(1000) * 0.5 + 3)))

In [1673]: plt.figure()
Out[1673]: <matplotlib.figure.Figure at 0x1de22750>

In [1674]: s.hist(normed=True, alpha=0.2)
Out[1674]: <matplotlib.axes.AxesSubplot at 0x1e7e7ad0>

In [1675]: s.plot(kind='kde')
Out[1675]: <matplotlib.axes.AxesSubplot at 0x1e7e7ad0>
```

See the plotting page for much more.

### 1.6.6 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.

### 1.6.7 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 replaced by `DatetimeIndex` but otherwise behaved identically. But, if you have code that converts `DateRange` or `Index` objects that used to contain `datetime.datetime` values to plain NumPy arrays, you may have bugs lurking with code using scalar values because you are handing control over to NumPy:

```python
In [1676]: import datetime

In [1677]: rng = date_range('1/1/2000', periods=10)
```
pandas's Timestamp object is a subclass of datetime.datetime that has nanosecond support (the nanosecond field store the nanosecond value between 0 and 999). It should substitute directly into any code that used datetime.datetime values before. Thus, I recommend not casting DatetimeIndex to regular NumPy arrays.

If you have code that requires an array of datetime.datetime objects, you have a couple of options. First, the asobject property of DatetimeIndex produces an array of Timestamp objects:

```
In [1683]: stamp_array = rng.asobject
```

To get an array of proper datetime.datetime objects, use the to_pydatetime method:

```
In [1686]: dt_array = rng.to_pydatetime()
```

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 [1689]: rng = date_range('1/1/2000', periods=10)

In [1690]: rng
Out[1690]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-10 00:00:00]
Length: 10, Freq: D, Timezone: None

In [1691]: np.asarray(rng)
Out[1691]:
array([1970-01-11 18:00:00, 1970-01-11 20:00:00, 1970-01-11 23:00:00,
     1970-01-11 00:00:00, 1970-01-11 24:00:00, 1970-01-11 48:00:00,
     1970-01-11 72:00:00, 1970-01-11 96:00:00, 1970-01-11 120:00:00,
     1970-01-11 144:00:00], dtype=datetime64[ns])

In [1692]: converted = np.asarray(rng, dtype=object)

In [1693]: converted[5]
Out[1693]: datetime.datetime(1970, 1, 11, 48, 0)

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.

1.7 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.

1.7.1 New features

- New fixed width file reader, read_fwf
- New scatter_matrix function for making a scatter plot matrix

```python
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

1.7.2 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 [1694]: series = Series(['Steve', np.nan, 'Joe'])

In [1695]: series == 'Steve'
Out[1695]:
0   True
1  False
2  False
dtype: bool

In [1696]: series != 'Steve'
Out[1696]:
0  False
1   True
2   True
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 [1697]: mask = series == 'Steve'

In [1698]: series[mask & series.notnull()]
Out[1698]:
0   Steve
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.

1.7.3 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:

```python
In [1699]: 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 [1700]: df
Out[1700]:
   A   B       C       D
0  foo  one  0.255678 -0.295328
1  bar  one  0.055213 -2.208596
2  foo  two  0.190798 -0.097122
3  bar  three  0.247394  0.289633
4  foo  two  0.453403  0.160891
5  bar  two  1.925709  0.929363
6  foo  one  0.714705  0.348722
7  foo  three  1.781358  0.352378
In [1701]: grouped = df.groupby('A')['C']
In [1702]: grouped.describe()
Out[1702]:
   A
bar  count  3.000000
   mean   0.742772
   std    1.028950
   min   -0.190798
   25%    0.151304
   50%    0.247394
   75%    1.086551
   max    1.925709
foo  count  5.000000
   mean   0.602869
   std    0.737247
   min   -0.190798
   25%    0.255678
   50%    0.453403
   75%    0.714705
   max    1.781358
dtype: float64
In [1703]: grouped.apply(lambda x: x.order()[-2:])  # top 2 values
Out[1703]:
   A
bar  3  0.247394
      5  1.925709
```

1.7. v.0.7.3 (April 12, 2012)
foo    6    0.714705
    7    1.781358
dtype: float64

1.8  v.0.7.2 (March 16, 2012)

This release targets bugs in 0.7.1, and adds a few minor features.

1.8.1  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)

1.8.2  Performance improvements

- Use khash for Series.value_counts, add raw function to algorithms.py (GH861)
- Intercept __builtin__.sum in groupby (GH885)

1.9  v.0.7.1 (February 29, 2012)

This release includes a few new features and addresses over a dozen bugs in 0.7.0.

1.9.1  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
1.9.2 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)

1.10 v.0.7.0 (February 9, 2012)

1.10.1 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)

```python
In [1704]: df = DataFrame(randn(10, 4))
In [1705]: df.apply(lambda x: x.describe())
```

```
Out[1705]:
       0         1         2         3
count 10.000000 10.000000 10.000000 10.000000
mean   0.467956 -0.271208  0.320479  0.255854
std    1.371729  0.759631  0.962485  0.661379
min   -2.055090 -1.349964 -0.905821 -0.904464
25%    -0.357215 -0.622232 -0.529872 -0.186021
50%    0.576651 -0.298812  0.317472  0.345715
75%    1.710590 -0.184047  1.031023  0.830109
max    2.061568  1.147814  1.641604  1.034401
```
- **Add** `reorder_levels` method to Series and DataFrame (PR534)
- **Add** dict-like `get` function to DataFrame and Panel (PR521)
- **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, PR552, others)
- **Add** attribute-based item access to `Panel` and add IPython completion (PR563)
- **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 (PR563)
• *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`.

**1.10.2 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:

```python
In [1706]: s = Series(randn(10), index=range(0, 20, 2))
In [1707]: s
Out[1707]:
0    0.910822
2    0.695714
4   -0.955386
6    0.359339
8   -0.189177
10  -1.168504
12  -1.381056
14   0.786651
16   0.288704
18   0.148544
dtype: float64
```

```python
In [1708]: s[0]
Out[1708]: 0.910822002253868
```

```python
In [1709]: s[2]
Out[1709]: 0.69571446395605785
```
In [1710]: s[4]
Out[1710]: -0.95538648457113173

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.iget_value(i)</td>
<td>Retrieve value stored at location i</td>
</tr>
<tr>
<td>Series.iget(i)</td>
<td>Alias for iget_value</td>
</tr>
<tr>
<td>DataFrame.irow(i)</td>
<td>Retrieve the i-th row</td>
</tr>
<tr>
<td>DataFrame.icol(j)</td>
<td>Retrieve the j-th column</td>
</tr>
<tr>
<td>DataFrame.iget_value(i, j)</td>
<td>Retrieve the value at row i and column j</td>
</tr>
</tbody>
</table>

### 1.10.3 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 [1711]: s = Series(randn(6), index=list('gmkaec'))

In [1712]: s
Out[1712]:
g   0.842702
m -1.876494
k -0.365497
a  -2.231289
e   0.716546
c  -0.958511
dtype: float64

Then this is OK:

In [1713]: s.ix[‘k’:’e’]
Out[1713]:
  k  -0.365497
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 [1714]: s2 = s.sort_index()

In [1715]: s2
Out[1715]:
a  -2.231289
c  -0.069151
e  0.716546
g  0.842702
k  -0.365497
m  -1.876494
dtype: float64

In [1716]: s2.ix['b':'h']
Out[1716]:
c  -0.069151
e  0.716546
g  0.842702
dtype: float64

1.10.4 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 [1717]: s = Series(randn(6), index=list('acegkm'))

In [1718]: s
Out[1718]:
a  -0.651033
c  -1.163455
e  1.627107
g  2.008883
k  -0.431064
m  -1.687776
dtype: float64

In [1719]: s[['m', 'a', 'c', 'e']]
Out[1719]:
m  -1.687776
a  -0.651033
c  -1.163455
e  1.627107
dtype: float64

In [1720]: s['b':'l']
In the case of integer indexes, the behavior will be exactly as before (shadowing `ndarray`):

```
In [1722]: s = Series(randn(6), index=range(0, 12, 2))
```

```
In [1723]: s[[4, 0, 2]]
```

```
Out[1723]:
4  -0.593879
0  -0.271860
2   1.101084
dtype: float64
```

If you wish to do indexing with sequences and slicing on an integer index with label semantics, use `ix`.

### 1.10.5 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)

### 1.10.6 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)

1.11 v.0.6.1 (December 13, 2011)

1.11.1 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 (PR435)
• 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 (PR453)
• 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 (PR482)
• Can pass DataFrame/DataFrame and DataFrame/Series to rolling_corr/rolling_cov (GH #462)
• MultiIndex.get_level_values can accept the level name
1.11.2 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)

1.12 v.0.6.0 (November 25, 2011)

1.12.1 New Features

- Added melt function to pandas.core.reshape
- Added level parameter to group by level in Series and DataFrame descriptive statistics (PR313)
- Added head and tail methods to Series, analogous to to DataFrame (PR296)
- 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 (PR321)
- 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, PR362)
- Added fast get_value and put_value methods to DataFrame (GH360)
- Added cov instance methods to Series and DataFrame (GH194, PR362)
- Added kind='bar' option to DataFrame.plot (PR348)
- Added idxmin and idxmax to Series and DataFrame (PR286)
- 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 (PR387)
- Added support for MaskedArray data in DataFrame, masked values converted to NaN (PR396)
• *Added* DataFrame.boxplot function (GH368)
• *Can* pass extra args, kwds to DataFrame.apply (GH376)
• *Implemented* 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)

### 1.12.2 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 (PR355)
• 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)

### 1.13 v0.5.0 (October 24, 2011)

#### 1.13.1 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. (PR233, GH230)
• **Implemented** Series.describe for Series containing objects (PR241)
• **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 (PR334)
• **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` (PR244)
• TODO: DOCS ABOUT TAKE METHODS

1.13.2 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

1.14 v0.4.3 through v0.4.1 (September 25 - October 9, 2011)

1.14.1 New Features

• Added Python 3 support using 2to3 (PR200)
• **Added** `name` attribute to `Series`, now prints as part of `Series.__repr__`
• **Added** instance methods `isnull` and `notnull` to `Series` (PR209, GH203)
• **Added** `Series.align` method for aligning two series with choice of join method (ENH56)
• **Added** method `get_level_values` to `MultiIndex` (IS188)
• 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` (PR146)
• `read_csv` can read multiple columns into a MultiIndex; `DataFrame`'s `to_csv` method writes out a corresponding MultiIndex (PR151)
• `DataFrame.rename` has a new `copy` parameter to `rename` a DataFrame in place (ENHed)
• Enable unstacking by name (PR142)
• Enable `sortlevel` to work by level (PR141)

1.14.2 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
You have the option to install an official release or to build the development version. If you choose to install from source and are running Windows, you will have to ensure that you have a compatible C compiler (MinGW or Visual Studio) installed. How-to install MinGW on Windows

2.1 Python version support

Officially Python 2.5 to 2.7 and Python 3.1+, although Python 3 support is less well tested. Python 2.4 support is being phased out since the userbase has shrunk significantly. Continuing Python 2.4 support will require either monetary development support or someone contributing to the project to maintain compatibility.

2.2 Binary installers

2.2.1 All platforms

Stable installers available on PyPI

Preliminary builds and installers on the Pandas download page.
## 2.2.2 Overview

<table>
<thead>
<tr>
<th>Platform</th>
<th>Distribution</th>
<th>Status</th>
<th>Download / Repository Link</th>
<th>Install method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>all</td>
<td>stable</td>
<td>All platforms</td>
<td>pip install pandas</td>
</tr>
<tr>
<td>Mac</td>
<td>all</td>
<td>stable</td>
<td>All platforms</td>
<td>pip install pandas</td>
</tr>
<tr>
<td>Linux</td>
<td>Debian</td>
<td>stable</td>
<td>official Debian repository</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>Linux</td>
<td>Debian &amp; Ubuntu</td>
<td>unstable (latest packages)</td>
<td>NeuroDebian</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>Linux</td>
<td>Ubuntu</td>
<td>stable</td>
<td>official Ubuntu repository</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>Linux</td>
<td>Ubuntu</td>
<td>unstable (daily builds)</td>
<td>PythonXY PPA; activate by: sudo add-apt-repository ppa:pythonxy/pythonxy-devel &amp;&amp; sudo apt-get update</td>
<td>sudo apt-get install python-pandas</td>
</tr>
<tr>
<td>Linux</td>
<td>OpenSuse &amp; Fedora</td>
<td>stable</td>
<td>OpenSuse Repository</td>
<td>zypper in python-pandas</td>
</tr>
</tbody>
</table>

## 2.3 Dependencies

- NumPy: 1.6.1 or higher
- python-dateutil 1.5
- pytz
  - Needed for time zone support

## 2.4 Optional dependencies

- Cython: Only necessary to build development version. Version 0.17.1 or higher.
- SciPy: miscellaneous statistical functions
- PyTables: necessary for HDF5-based storage
- matplotlib: for plotting
- statsmodels
  - Needed for parts of pandas.stats
- openpyxl, xlr/xlw
  - openpyxl version 1.6.1 or higher
  - Needed for Excel I/O
Note: Without the optional dependencies, many useful features will not work. Hence, it is highly recommended that you install these. A packaged distribution like the Enthought Python Distribution may be worth considering.

2.5 Installing from source

Note: Installing from the git repository requires a recent installation of Cython as the cythonized C sources are no longer checked into source control. Released source distributions will contain the built C files. I recommend installing the latest Cython via `easy_install -U Cython`

The source code is hosted at http://github.com/pydata/pandas, it can be checked out using git and compiled / installed like so:

git clone git://github.com/pydata/pandas.git
cd pandas
python setup.py install

Make sure you have Cython installed when installing from the repository, rather then a tarball or pypi.

On Windows, I suggest installing the MinGW compiler suite following the directions linked to above. Once configured property, run the following on the command line:

`python setup.py build --compiler=mingw32`
`python setup.py install`

Note that you will not be able to import pandas if you open an interpreter in the source directory unless you build the C extensions in place:

`python setup.py build_ext --inplace`

The most recent version of MinGW (any installer dated after 2011-08-03) has removed the `-mno-cygwin` option but Distutils has not yet been updated to reflect that. Thus, you may run into an error like “unrecognized command line option ‘-mno-cygwin’”. Until the bug is fixed in Distutils, you may need to install a slightly older version of MinGW (2011-08-02 installer).

2.6 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 you have all of the dependencies, soft and hard, installed), make sure you have `nose` and run:

```
$ nosetests pandas
```

```
..........................................................................
...............................S.............................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
..........................................................................
```

2.5. Installing from source
Ran 818 tests in 21.631s

OK (SKIP=2)
FREQUENTLY ASKED QUESTIONS
(FAQ)

Pandas is a powerful tool and already has a plethora of data manipulation operations implemented, most of them are very fast as well. It’s very possible however that certain functionality that would make your life easier is missing. In that case you have several options:

1. Open an issue on Github, explain your need and the sort of functionality you would like to see implemented.
2. Fork the repo, Implement the functionality yourself and open a PR on Github.
3. Write a method that performs the operation you are interested in and Monkey-patch the pandas class as part of your IPython profile startup or PYTHONSTARTUP file.

For example, here is an example of adding an `just_foo_cols()` method to the dataframe class:

```python
In [452]: import pandas as pd

In [453]: def just_foo_cols(self):
   .....:     return [x for x in self.columns if 'foo' in x]
   .....:

In [454]: pd.DataFrame.just_foo_cols = just_foo_cols # monkey-patch the DataFrame class

In [455]: df = pd.DataFrame([range(4),columns= ['A','foo','foozball','bar']])

In [456]: df.just_foo_cols()
Out[456]: ['foo', 'foozball']

In [457]: del pd.DataFrame.just_foo_cols # you can also remove the new method
```

Monkey-patching is usually frowned upon because it makes your code less portable and can cause subtle bugs in some circumstances. Monkey-patching existing methods is usually a bad idea in that respect. When used with proper care, however, it’s a very useful tool to have.

### 3.1 Migrating from scikits.timeseries to pandas >= 0.8.0

Starting with pandas 0.8.0, users of scikits.timeseries should have all of the features that they need to migrate their code to use pandas. Portions of the scikits.timeseries codebase for implementing calendar logic and timespan frequency
conversions (but not resampling, that has all been implemented from scratch from the ground up) have been ported to the pandas codebase.

The scikits.timeseries notions of Date and DateArray are responsible for implementing calendar logic:

In [16]: dt = ts.Date('Q', '1984Q3')

# sic
In [17]: dt
Out[17]: <Q-DEC : 1984Q1>

In [18]: dt.asfreq('D', 'start')
Out[18]: <D : 01-Jan-1984>

In [19]: dt.asfreq('D', 'end')
Out[19]: <D : 31-Mar-1984>

In [20]: dt + 3
Out[20]: <Q-DEC : 1984Q4>

Date and DateArray from scikits.timeseries have been reincarnated in pandas Period and PeriodIndex:

In [458]: pnow('D')  # scikits.timeseries.now()
Out[458]: Period('2013-04-23', 'D')

In [459]: Period(year=2007, month=3, day=15, freq='D')
Out[459]: Period('2007-03-15', 'D')

In [460]: p = Period('1984Q3')

In [461]: p
Out[461]: Period('1984Q3', 'Q-DEC')

In [462]: p.asfreq('D', 'start')
Out[462]: Period('1984-07-01', 'D')

In [463]: p.asfreq('D', 'end')
Out[463]: Period('1984-09-30', 'D')

In [464]: (p + 3).asfreq('T') + 6 * 60 + 30
Out[464]: Period('1985-07-01 06:29', 'T')

In [465]: rng = period_range('1990', '2010', freq='A')

In [466]: rng
Out[466]: <class 'pandas.tseries.period.PeriodIndex'>
freq: A-DEC
[1990, ..., 2010]
length: 21

In [467]: rng.asfreq('B', 'end') - 3
Out[467]: <class 'pandas.tseries.period.PeriodIndex'>
freq: B
[1990-12-26, ..., 2010-12-28]
length: 21
### 3.1.1 PeriodIndex / DateArray properties and functions

The scikits.timeseries `DateArray` had a number of information properties. Here are the pandas equivalents:

<table>
<thead>
<tr>
<th>scikits.timeseries</th>
<th>pandas</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_steps</td>
<td><code>np.diff(idx.values)</code></td>
<td></td>
</tr>
<tr>
<td>has_missing_dates</td>
<td><code>not idx.is_full</code></td>
<td></td>
</tr>
<tr>
<td>is_full</td>
<td><code>idx.is_full</code></td>
<td></td>
</tr>
<tr>
<td>is_valid</td>
<td><code>idx.is_monotonic and idx.is_unique</code></td>
<td></td>
</tr>
<tr>
<td>is_chronological</td>
<td><code>is_monotonic</code></td>
<td></td>
</tr>
<tr>
<td>arr.sort_chronologically</td>
<td><code>idx.order()</code></td>
<td></td>
</tr>
</tbody>
</table>

### 3.1.2 Frequency conversion

Frequency conversion is implemented using the `resample` method on `TimeSeries` and `DataFrame` objects (multiple time series). `resample` also works on panels (3D). Here is some code that resamples daily data to monthly with scikits.timeseries:

**In [468]:** `import scikits.timeseries as ts`

**In [469]:**
```python
data = ts.time_series(np.random.randn(50), start_date='Jan-2000', freq='M')
```

**In [470]:**
```python
data
```
```
timeseries([-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 0.4047 0.577 -1.715 -1.0393 -0.3706 -1.1579 -1.3443 0.8449 1.0758 -0.109 1.6436 -1.4694 0.357 -0.6746 -1.7769 -0.9689 -1.2945 0.4137 0.2767 -0.472 -0.014 -0.3625 -0.0062 -0.9231 0.8957 0.8052],
   dates = [Jan-2013 ... Feb-2017],
   freq = M)
```

**In [471]:**
```python
data.convert('A', func=np.mean)
```
```
timeseries([-0.394509620575 -0.24462765889 -0.221632512996 -0.453772693384 0.8504806638],
   dates = [2013 ... 2017],
   freq = A-DEC)
```

Here is the equivalent pandas code:

**In [472]:**
```python
rng = period_range('Jan-2000', periods=50, freq='M')
```

**In [473]:**
```python
data = Series(np.random.randn(50), index=rng)
```

**In [474]:**
```python
data
```
```
2000-01   -1.206412
```

---

3.1. Migrating from scikits.timeseries to pandas >= 0.8.0
In [475]:
data.resample('A', how=np.mean)
Out[475]:
2000  0.166630
2001 -0.114581
2002 -0.205961
2003 -0.235802
2004 -1.284876

Freq: M, dtype: float64
3.1.3 Plotting

Much of the plotting functionality of scikits.timeseries has been ported and adopted to pandas’s data structures. For example:

In [476]: rng = period_range('1987Q2', periods=10, freq='Q-DEC')

In [477]: data = Series(np.random.randn(10), index=rng)

In [478]: plt.figure(); data.plot()

Out[478]: <matplotlib.axes.AxesSubplot at 0x9bce750>

3.1.4 Converting to and from period format

Use the `to_timestamp` and `to_period` instance methods.

3.1.5 Treatment of missing data

Unlike scikits.timeseries, pandas data structures are not based on NumPy’s `MaskedArray` object. Missing data is represented as `NaN` in numerical arrays and either as `None` or `NaN` in non-numerical arrays. Implementing a version of pandas’s data structures that use `MaskedArray` is possible but would require the involvement of a dedicated maintainer. Active pandas developers are not interested in this.

3.1.6 Resampling with timestamps and periods

`resample` has a `kind` argument which allows you to resample time series with a `DatetimeIndex` to `PeriodIndex`:

3.1. Migrating from scikits.timeseries to pandas >= 0.8.0
In [479]: rng = date_range('1/1/2000', periods=200, freq='D')

In [480]: data = Series(np.random.randn(200), index=rng)

In [481]: data[:10]
Out[481]:
2000-01-01 -0.487602
2000-01-02 -0.082240
2000-01-03 -2.182937
2000-01-04  0.380396
2000-01-05  0.084844
2000-01-06  0.432390
2000-01-07  1.519970
2000-01-08 -0.493662
2000-01-09  0.600178
2000-01-10  0.274230
Freq: D, dtype: float64

In [482]: data.index
Out[482]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-07-18 00:00:00]
Length: 200, Freq: D, Timezone: None

In [483]: data.resample('M', kind='period')
Out[483]:
2000-01  0.163775
2000-02  0.026549
2000-03 -0.089563
2000-04 -0.079405
2000-05  0.160348
2000-06  0.101725
2000-07 -0.708770
Freq: M, dtype: float64

Similarly, resampling from periods to timestamps is possible with an optional interval ('start' or 'end') convention:

In [484]: rng = period_range('Jan-2000', periods=50, freq='M')

In [485]: data = Series(np.random.randn(50), index=rng)

In [486]: resampled = data.resample('A', kind='timestamp', convention='end')

In [487]: resampled.index
Out[487]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-12-31 00:00:00, ..., 2004-12-31 00:00:00]
Length: 5, Freq: A-DEC, Timezone: None
pandas consists of the following things

- A set of labeled array data structures, the primary of which are Series/TimeSeries 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.)
- Static and moving window linear and panel regression

4.1 Data structures at a glance

<table>
<thead>
<tr>
<th>Dimensions</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Series</td>
<td>1D labeled homogeneously-typed array</td>
</tr>
<tr>
<td>1</td>
<td>Time-Series</td>
<td>Series with index containing datetimes</td>
</tr>
<tr>
<td>2</td>
<td>DataFrame</td>
<td>General 2D labeled, size-mutable tabular structure with potentially</td>
</tr>
<tr>
<td></td>
<td></td>
<td>heterogeneously-typed columns</td>
</tr>
<tr>
<td>3</td>
<td>Panel</td>
<td>General 3D labeled, also size-mutable array</td>
</tr>
</tbody>
</table>

4.1.1 Why more than 1 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 Panel is a container for DataFrame objects. 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-contiguousness 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. And iterating through the columns of the DataFrame thus results in more readable code:

```python
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, though, 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.

Longer discussions occur on the developer mailing list, and commercial support inquiries for Lambda Foundry should be sent to: support@lambdafoundry.com

### 4.4 Credits

pandas development began at AQR Capital Management in April 2008. It was open-sourced at the end of 2009. AQR continued to provide resources for development through the end of 2011, and continues to contribute bug reports today.

Since January 2012, Lambda Foundry, has been providing development resources, as well as commercial support, training, and consulting for pandas.

pandas is only made possible by a group of people around the world like you who have contributed new code, bug reports, fixes, comments and ideas. A complete list can be found on Github.

### 4.5 Development Team

pandas is a part of the PyData project. The PyData Development Team is a collection of developers focused on the improvement of Python’s data libraries. The core team that coordinates development can be found on Github. If you’re interested in contributing, please visit the project website.

### 4.6 License
pandas is distributed under a 3-clause ("Simplified" or "New") BSD license. Parts of NumPy, SciPy, numpydoc, bottleneck, which all have BSD-compatible licenses, are included. Their licenses follow the pandas license.

pandas license
==============

Copyright (c) 2011-2012, Lambda Foundry, Inc. and PyData Development Team
All rights reserved.

Copyright (c) 2008-2011 AQR Capital Management, LLC
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 any 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 HOLDER 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 OWNER 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.

About the Copyright Holders
===========================

AQR Capital Management began pandas development in 2008. Development was led by Wes McKinney. AQR released the source under this license in 2009. Wes is now an employee of Lambda Foundry, and remains the pandas project lead.

The PyData Development Team is the collection of developers of the PyData project. This includes all of the PyData sub-projects, including pandas. The core team that coordinates development on GitHub can be found here: http://github.com/pydata.
Full credits for pandas contributors can be found in the documentation.

Our Copyright Policy
====================

PyData uses a shared copyright model. Each contributor maintains copyright over their contributions to PyData. However, it is important to note that these contributions are typically only changes to the repositories. Thus, the PyData source code, in its entirety, is not the copyright of any single person or institution. Instead, it is the collective copyright of the entire PyData Development Team. If individual contributors want to maintain a record of what changes/contributions they have specific copyright on, they should indicate their copyright in the commit message of the change when they commit the change to one of the PyData repositories.

With this in mind, the following banner should be used in any source code file to indicate the copyright and license terms:

#-------------------------------------------------
# Copyright (c) 2012, PyData Development Team
# All rights reserved.
# Distributed under the terms of the BSD Simplified License.
# The full license is in the LICENSE file, distributed with this software.
#-------------------------------------------------

Other licenses can be found in the LICENSES directory.
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 [305]: import numpy as np
# will use a lot in examples
In [306]: randn = np.random.randn
In [307]: from pandas import *
```

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.

When using pandas, we recommend the following import convention:

```
import pandas as pd
```

### 5.1 Series

Series is a one-dimensional labeled array (technically a subclass of ndarray) 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 = 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 

\[0, \ldots, \text{len(data)} - 1\].
In [308]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [309]: s
Out[309]:
a    0.314
b   -0.002
c    0.072
d    0.893
e    0.681
dtype: float64

In [310]: s.index
Out[310]: Index([a, b, c, d, e], dtype=object)

In [311]: Series(randn(5))
Out[311]:
0   -0.340
1    0.215
2   -0.078
3   -0.178
4    0.491
dtype: float64

Note: Starting in v0.8.0, pandas supports non-unique index values. In previous version, if the index values are not unique an exception will not be raised immediately, but attempting any operation involving the index will later result in an exception. In other words, the Index object containing the labels “lazily” checks whether the values are unique. 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

If data is a dict, if index is passed the values in data corresponding to the labels in the index will be pulled out. Otherwise, an index will be constructed from the sorted keys of the dict, if possible.

In [312]: d = {'a' : 0., 'b' : 1., 'c' : 2.}

In [313]: Series(d)
Out[313]:
a     0
b     1
c     2
dtype: float64

In [314]: Series(d, index=['b', 'c', 'd', 'a'])
Out[314]:
b    1
  c    2
  d   NaN
  a    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 [315]: Series(5., index=['a', 'b', 'c', 'd', 'e'])
Out[315]:
a   5
b   5
c   5
d   5
e   5
dtype: float64

5.1.1 Series is ndarray-like

As a subclass of ndarray, Series is a valid argument to most NumPy functions and behaves similarly to a NumPy array. However, things like slicing also slice the index.

In [316]: s[0]
Out[316]: 0.31422552353417077

In [317]: s[:3]
Out[317]:
a   0.314
b  -0.002
c   0.072
dtype: float64

In [318]: s[s > s.median()]
Out[318]:
d   0.893
e   0.681
dtype: float64

In [319]: s[[4, 3, 1]]
Out[319]:
e   0.681
d   0.893
b  -0.002
dtype: float64

In [320]: np.exp(s)
Out[320]:
a   1.369
b   0.998
c   1.074
d   2.441
e   1.975
dtype: float64

We will address array-based indexing in a separate section.

5.1.2 Series is dict-like

A Series is like a fixed-size dict in that you can get and set values by index label:

In [321]: s['a']
Out[321]: 0.31422552353417077

In [322]: s['e'] = 12.
In [323]: s
Out[323]:
a    0.314
b   -0.002
c     0.072
d     0.893
e    12.000
dtype: float64

In [324]: 'e' in s
Out[324]: True

In [325]: 'f' in s
Out[325]: False

If a label is not contained, an exception is raised:

```python
>>> s['f']
KeyError: 'f'
```

Using the `get` method, a missing label will return None or specified default:

```python
In [326]: s.get('f')
In [327]: s.get('f', np.nan)
Out[327]: nan
```

### 5.1.3 Vectorized operations and label alignment with Series

When doing data analysis, as with raw NumPy arrays looping through Series value-by-value is usually not necessary. Series can be also be passed into most NumPy methods expecting an ndarray.

```python
In [328]: s + s
Out[328]:
a    0.628
b   -0.003
c     0.144
d     1.785
e    24.000
dtype: float64

In [329]: s * 2
Out[329]:
a    0.628
b   -0.003
c     0.144
d     1.785
e    24.000
dtype: float64

In [330]: np.exp(s)
Out[330]:
a    1.369
b    0.998
c    1.074
d    2.441
```
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.

```python
In [331]: s[1:] + s[:-1]
Out[331]:
a    NaN
b   -0.003
c    0.144
d    1.785
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.

### 5.1.4 Name attribute

Series can also have a `name` attribute:

```python
In [332]: s = Series(np.random.randn(5), name='something')
```

```python
In [333]: s
```

```
Out[333]:
0  -1.360
1   1.592
2   1.007
3   0.698
4  -1.891
Name: something, dtype: float64
```

```python
In [334]: s.name
```

```
Out[334]: 'something'
```

The Series `name` will be assigned automatically in many cases, in particular when taking 1D slices of DataFrame as you will see below.

### 5.2 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.

### 5.2.1 From dict of Series or dicts

The result **index** will be the **union** of the indexes of the various Series. If there are any nested dicts, these will be first converted to Series. If no columns are passed, the columns will be the sorted list of dict keys.

```
In [335]: d = {'one' : Series([1., 2., 3.], index=['a', 'b', 'c']),
       'two' : Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])
       ....:
       'three' : Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])
       ....:

In [336]: df = DataFrame(d)

In [337]: df
Out[337]:
   one  two
  a  1  1
  b  2  2
  c  3  3
  d  NaN 4

In [338]: DataFrame(d, index=['d', 'b', 'a'])
Out[338]:
   one  two
  d  NaN 4
  b  2  2
  a  1  1

In [339]: DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
Out[339]:
   two  three
  d  4  NaN
  b  2  NaN
  a  1  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 [340]: df.index
Out[340]: Index(['a', 'b', 'c', 'd'], dtype=object)
```
In [341]: df.columns
Out[341]: Index([one, two], dtype=object)

5.2.2 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 [342]: d = {'one': [1., 2., 3., 4.],
       ...:   'two': [4., 3., 2., 1.]}  
      ...:

In [343]: DataFrame(d)
Out[343]:
     one two
0 1   4
1 2   3
2 3   2
3 4   1

In [344]: DataFrame(d, index=['a', 'b', 'c', 'd'])
Out[344]:
     one two
a 1   4
b 2   3
c 3   2
d 4   1

5.2.3 From structured or record array

This case is handled identically to a dict of arrays.

In [345]: data = np.zeros((2,), dtype=[('A', 'i4'), ('B', 'f4'), ('C', 'a10')])

In [346]: data[:] = [(1, 2., 'Hello'), (2, 3., 'World')]

In [347]: DataFrame(data)
Out[347]:
      A  B    C
0  1  2  Hello
1  2  3     World

In [348]: DataFrame(data, index=['first', 'second'])
Out[348]:
      A  B    C
first 1  2  Hello
second 2  3     World

In [349]: DataFrame(data, columns=['C', 'A', 'B'])
Out[349]:
      C  A  B
0    Hello  1  2
1     World  2  3

Note: DataFrame is not intended to work exactly like a 2-dimensional NumPy ndarray.
5.2.4 From a list of dicts

In [350]: data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]

In [351]: DataFrame(data2)
Out[351]:
   a   b  c
0  1   2 NaN
1  5  10 20

In [352]: DataFrame(data2, index=['first', 'second'])
Out[352]:
       a   b  c
first  1   2 NaN
second 5  10 20

In [353]: DataFrame(data2, columns=['a', 'b'])
Out[353]:
   a   b
0  1   2
1  5  10

5.2.5 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, use np.nan for those values which are missing. Alternatively, you may pass a numpy.MaskedArray as the data argument to the DataFrame constructor, and its masked entries will be considered missing.

5.2.6 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. DataFrame.from_records

DataFrame.from_records takes a list of tuples or an ndarray with structured dtype. Works analogously to the normal DataFrame constructor, except that index maybe be a specific field of the structured dtype to use as the index. For example:

In [354]: data
Out[354]:
  array([(1, 2.0, 'Hello'), (2, 3.0, 'World')],
        dtype=[('A', '<i4'), ('B', '<f4'), ('C', '|S10')])

In [355]: DataFrame.from_records(data, index='C')
Out[355]:
   A   B
C

66 Chapter 5. Intro to Data Structures
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

Hello 1 2
World 2 3

**DataFrame.from_items**

`DataFrame.from_items` works analogously to the form of the `dict` constructor that takes a sequence of `(key, value)` pairs, where the keys are column (or row, in the case of `orient='index'`) names, and the value are the column values (or row values). This can be useful for constructing a DataFrame with the columns in a particular order without having to pass an explicit list of columns:

```
In [356]: DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])
```

```
Out[356]:
     A  B
   0  1  4
   1  2  5
   2  3  6
```

If you pass `orient='index'`, the keys will be the row labels. But in this case you must also pass the desired column names:

```
In [357]: DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
  .....:   orient='index', columns=['one', 'two', 'three'])
```

```
Out[357]:
      one  two  three
   A    1    2    3
   B    4    5    6
```

### 5.2.7 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 [358]: df['one']
```

```
Out[358]:
   a    1
   b    2
   c    3
   d  NaN
Name: one, dtype: float64
```

```
In [359]: df['three'] = df['one'] * df['two']

In [360]: df['flag'] = df['one'] > 2
```

```
In [361]: df
```

```
Out[361]:
      one  two  three  flag
   a    1    1    1   False
   b    2    2    4  False
   c    3    3    9   True
   d  NaN    4   NaN  False
```

Columns can be deleted or popped like with a dict:

```
In [362]: del df['two']

In [363]: three = df.pop('three')
```

---

5.2. DataFrame 67
In [364]: df
Out[364]:
    one  flag
  a    1  False
  b    2  False
  c    3   True
  d  NaN  False

When inserting a scalar value, it will naturally be propagated to fill the column:

In [365]: df['foo'] = 'bar'
In [366]: df
Out[366]:
    one  flag  foo
  a    1  False  bar
  b    2  False  bar
  c    3   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 [367]: df['one_trunc'] = df['one'][:2]
In [368]: df
Out[368]:
    one  flag  foo  one_trunc
  a    1  False  bar   1
  b    2  False  bar   2
  c    3   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 [369]: df.insert(1, 'bar', df['one'])
In [370]: df
Out[370]:
    one  bar  flag  foo  one_trunc
  a    1    1  False  bar   1
  b    2    2  False  bar   2
  c    3    3   True  bar  NaN
  d  NaN  NaN  False  bar  NaN

5.2.8 Indexing / Selection

The basics of indexing are as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select column</td>
<td>df[col]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by label</td>
<td>df.xs(label) or df.ix[label]</td>
<td>Series</td>
</tr>
<tr>
<td>Select row by location (int)</td>
<td>df.ix[loc]</td>
<td>Series</td>
</tr>
<tr>
<td>Slice rows</td>
<td>df[5:10]</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Select rows by boolean vector</td>
<td>df[bool_vec]</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>
Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [371]: df.xs('b')
Out[371]:
one  2
bar  2
flag  False
foo  bar
one_trunc  2
Name: b, dtype: object
```

```
In [372]: df.ix[2]
Out[372]:
one  3
bar  3
flag  True
foo  bar
one_trunc  NaN
Name: c, dtype: object
```

Note if a DataFrame contains columns of multiple dtypes, the dtype of the row will be chosen to accommodate all of the data types (dtype=object is the most general).

For a more exhaustive treatment of more sophisticated label-based indexing and slicing, see the section on indexing. We will address the fundamentals of reindexing / conforming to new sets of labels in the section on reindexing.

### 5.2.9 Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on both the columns and the index (row labels). Again, the resulting object will have the union of the column and row labels.

```
In [373]: df = DataFrame(randn(10, 4), columns=['A', 'B', 'C', 'D'])
In [374]: df2 = DataFrame(randn(7, 3), columns=['A', 'B', 'C'])
In [375]: df + df2
Out[375]:
     A    B   C    D
0  0.229  1.547 -1.499    NaN
1  0.121 -0.234  0.133 -0.958
2 -0.561 -1.550  0.643    NaN
3 -0.263  1.071 -0.060    NaN
4 -2.588 -0.752 -1.227    NaN
5  0.628 -0.095 -3.236    NaN
6  0.983 -0.823 -0.720    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 [376]: df - df.ix[0]
Out[376]:
     A    B    C    D
0  0.000  0.000  0.000  0.000
1  0.879 -2.485  0.133 -0.958
2  0.246 -1.482  0.106  0.685
```
In the special case of working with time series data, if the Series is a TimeSeries (which it will be automatically if the index contains datetime objects), and the DataFrame index also contains dates, the broadcasting will be column-wise:

```python
In [377]: index = date_range('1/1/2000', periods=8)

In [378]: df = DataFrame(randn(8, 3), index=index, columns=['A', 'B', 'C'])

In [379]: df
Out[379]:
     A     B     C
2000-01-01  0.302  1.113 -0.543
2000-01-02 -2.696  0.431 -0.431
2000-01-03  1.667  0.717 -0.920
2000-01-04 -0.025  0.069  0.602
2000-01-05  0.867  0.093 -2.607
2000-01-06  0.309 -0.548 -2.045
2000-01-07 -1.666 -1.440  1.326
2000-01-08  0.222  1.841  1.165

In [380]: type(df['A'])
Out[380]: pandas.core.series.TimeSeries

In [381]: df - df['A']
Out[381]:
     A     B     C
2000-01-01  0  0.811 -0.845
2000-01-02  0  3.127  2.264
2000-01-03  0 -0.950 -2.586
2000-01-04  0  0.094  0.626
2000-01-05  0 -0.774 -3.474
2000-01-06  0 -0.858 -2.354
2000-01-07  0  0.226  2.993
2000-01-08  0  1.619  0.943

Technical purity aside, this case is so common in practice that supporting the special case is preferable to the alternative of forcing the user to transpose and do column-based alignment like so:

```python
In [382]: (df.T - df['A']).T
Out[382]:
     A     B     C
2000-01-01  0  0.811 -0.845
2000-01-02  0  3.127  2.264
2000-01-03  0 -0.950 -2.586
2000-01-04  0  0.094  0.626
2000-01-05  0 -0.774 -3.474
2000-01-06  0 -0.858 -2.354
2000-01-07  0  0.226  2.993
2000-01-08  0  1.619  0.943
```
For explicit control over the matching and broadcasting behavior, see the section on flexible binary operations.

Operations with scalars are just as you would expect:

In [383]: df * 5 + 2
Out[383]:
   A    B    C
0 2000-01-01 3.510 7.563 -0.714
1 2000-01-02 -11.478 4.156 -0.155
2 2000-01-03 10.333 5.583 -2.599
3 2000-01-04 1.877 2.345 5.009
4 2000-01-05 6.333 2.465 -11.034
5 2000-01-06 3.547 -0.742 -8.224
6 2000-01-07 -6.331 -5.199 8.632
7 2000-01-08 3.108 11.205 7.826

In [384]: 1 / df
Out[384]:
   A    B    C
0 2000-01-01 3.312 0.899 -1.842
1 2000-01-02 -0.371 2.319 -2.320
2 2000-01-03 0.600 1.395 -1.087
4 2000-01-05 1.154 10.763 -0.384
5 2000-01-06 3.233 -1.823 -0.489
6 2000-01-07 -0.600 -0.695 0.754
7 2000-01-08 4.511 0.543 0.858

In [385]: df ** 4
Out[385]:
   A    B    C
0 2000-01-01 8.312e-03 1.532e+00 0.087
1 2000-01-02 5.279e+01 3.460e-02 0.035
2 2000-01-03 7.715e+00 2.638e-01 0.716
3 2000-01-04 3.659e-07 2.266e-05 0.131
4 2000-01-05 5.640e-01 7.452e-05 46.184
5 2000-01-06 9.152e-03 9.045e-05 17.482
6 2000-01-07 7.709e+00 4.297e+00 3.095
7 2000-01-08 2.415e-03 1.149e+01 1.843

Boolean operators work as well:

In [386]: df1 = DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)
In [387]: df2 = DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
In [388]: df1 & df2
Out[388]:
   a    b
0  False  False
1   True   True
2   True  False

In [389]: df1 | df2
Out[389]:
   a    b
0   True   True
1   True   True
2   True   True
In [390]: df1 ^ df2
Out[390]:
   a    b
0  True  True
1   True  False
2  False  True

In [391]: -df1
Out[391]:
   a    b
0  False  True
1   True  False
2  False  False

5.2.10 Transposing

To transpose, access the T attribute (also the transpose function), similar to an ndarray:

# only show the first 5 rows
In [392]: df[:5].T
Out[392]:
A     0.302    -2.696     1.667    -0.025     0.867
B     1.113     0.431     0.717     0.069     0.093
C    -0.543    -0.431    -0.920     0.602    -2.607

5.2.11 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 [393]: np.exp(df)
Out[393]:
     A    B    C
2000-01-01  1.352  3.042  0.581
2000-01-02  0.068  1.539  0.650
2000-01-03  5.294  2.048  0.399
2000-01-04  0.976  1.071  1.825
2000-01-05  2.379  1.097  0.074
2000-01-06  1.362  0.578  0.129
2000-01-07  0.189  0.237  3.767
2000-01-08  1.248  6.303  3.206

In [394]: np.asarray(df)
Out[394]:
array([[ 0.3019,  1.1125, -0.5428],
       [-2.6955,  0.4313, -0.4311],
       [ 1.6666,  0.7167, -0.9197],
       [-0.0246,  0.069 ,  0.6018],
       [ 0.8666,  0.0929, -2.6069],
       [ 0.3093, -0.5484, -2.0448],
       [ 1.6663, -1.4398,  1.3264],
       [ 0.2217,  1.841 ,  1.1651]])

The dot method on DataFrame implements matrix multiplication:
In [395]: df.T.dot(df)
Out[395]:
   A   B  C
A  13.8  3.08 -5.39
B   3.08 7.714 -0.293
C -5.393   -0.293 15.782

Similarly, the dot method on Series implements dot product:

In [396]: s1 = Series(np.arange(5,10))
In [397]: s1.dot(s1)
Out[397]: 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.

5.2.12 Console display

For very large DataFrame objects, only a summary will be printed to the console (here I am reading a CSV version of the *baseball* dataset from the *plyr* R package):

In [398]: baseball = read_csv('data/baseball.csv')
In [399]:
print baseball
<class 'pandas.core.frame.DataFrame'>
Int64Index: 100 entries, 88641 to 89534
Data columns:
id 100 non-null values
year 100 non-null values
stint 100 non-null values
team 100 non-null values
lg 100 non-null values
g 100 non-null values
ab 100 non-null values
r 100 non-null values
h 100 non-null values
X2b 100 non-null values
X3b 100 non-null values
hr 100 non-null values
rbi 100 non-null values
sb 100 non-null values
cs 100 non-null values
bb 100 non-null values
so 100 non-null values
ibb 100 non-null values
hbp 100 non-null values
sh 100 non-null values
sf 100 non-null values
gidp 100 non-null values
dtypes: float64(9), int64(10), object(3)

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 [400]:
print baseball.ix[-20:, :12].to_string()
iid  year  stint  team   lg  g  ab  r  h  X2b  X3b  hr
88641 womacto01 2006 2 CHN NL 19 50 6 14 1 0 1
<table>
<thead>
<tr>
<th>Name</th>
<th>Team</th>
<th>Position</th>
<th>AGE</th>
<th>GS</th>
<th>Starts</th>
<th>Hrs</th>
<th>Games</th>
<th>Losses</th>
<th>Win%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schilcu01</td>
<td>BOS</td>
<td>AL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Helliri01</td>
<td>MIL</td>
<td>NL</td>
<td></td>
<td>6</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jchnsra05</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>58</td>
<td>139</td>
<td>6</td>
<td>105</td>
<td>21</td>
<td>12.6</td>
</tr>
<tr>
<td>Seleea01</td>
<td>LAN</td>
<td>NL</td>
<td></td>
<td>6</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Helliri01</td>
<td>MIL</td>
<td>NL</td>
<td></td>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>42</td>
<td>153</td>
<td>6</td>
<td>86</td>
<td>139</td>
<td>2.15</td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>50</td>
<td>150</td>
<td>10</td>
<td>40</td>
<td>15.1</td>
<td>3</td>
</tr>
<tr>
<td>Selna01</td>
<td>LAN</td>
<td>NL</td>
<td></td>
<td>7</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Johnsi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>4</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>7</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>3</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finest01</td>
<td>SFN</td>
<td>NL</td>
<td></td>
<td>1</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gonzalu01</td>
<td>ARI</td>
<td>NL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Myersmi01</td>
<td>NYA</td>
<td>AL</td>
<td></td>
<td>2</td>
<td></td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Chapter 5. Intro to Data Structures
New since 0.10.0, wide DataFrames will now be printed across multiple rows by default:

```
In [401]: DataFrame(randn(3, 12))
Out[401]:
     0  1  2  3  4  5  6  7  8  9 10 11
0  0.059323 -1.408245 -0.382548 0.038077 -0.282633 0.132684 -0.052227
1  0.091827 -0.068166 -0.758682 0.564969 0.137850 0.132345 -0.193399
2 -1.130313 -1.020858 -0.677853 1.515210 0.225588 0.198211 -0.102505
```

You can change how much to print on a single row by setting the `line_width` option:

```
In [402]: set_option('line_width', 40)  # default is 80
```
In [403]: DataFrame(randn(3, 12))
Out[403]:
    0  1  2  3  
0 -0.970022 -1.127997 -0.384526 -0.492429
1  1.295440  0.027006  0.863536  0.189023
2 -0.822538 -1.590312 -0.061405  0.400325
    4  5  6  7  
0 -1.779882 -0.391166  0.575903 -1.343193
1 -0.912154  0.946960 -0.257288  0.695208
2  1.511027  0.289143  0.349037  1.998562
    8  9 10 11
0  1.646841  0.462269  1.078574  0.883532
1  0.915200 -1.052414 -0.910945 -0.174453
2  1.056844 -0.077851 -0.057005  0.626302

You can also disable this feature via the expand_frame_repr option:

In [404]: set_option('expand_frame_repr', False)

In [405]: DataFrame(randn(3, 12))
Out[405]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 3 entries, 0 to 2
Data columns:
 0 3 non-null values
 1 3 non-null values
 2 3 non-null values
 3 3 non-null values
 4 3 non-null values
 5 3 non-null values
 6 3 non-null values
 7 3 non-null values
 8 3 non-null values
 9 3 non-null values
10 3 non-null values
11 3 non-null values
dtypes: float64(12)

5.2.13 DataFrame column types

The four main types stored in pandas objects are float, int, boolean, and object. A convenient dtypes attribute return a Series with the data type of each column:

In [406]: baseball.dtypes
Out[406]:
id object
year int64
stint int64
team object
lg object
g int64
ab int64
r int64
h int64
X2b int64
X3b int64
hr int64
rbi    float64
sb    float64
cs    float64
bb    int64
so    float64
ibb    float64
hbp    float64
sh    float64
sf    float64
gidp    float64
dtype: object

The related method `get_dtype_counts` will return the number of columns of each type:

In [407]: baseball.get_dtype_counts()
Out[407]:
float64    9
int64     10
object     3
dtype: int64

5.2.14 DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like attributes:

In [408]: df = DataFrame({'foo1': np.random.randn(5),
                      'foo2': np.random.randn(5)})

In [409]: df
Out[409]:
       foo1    foo2
0 -0.868315 -0.502919
1 -2.677551 -0.825049
2 -1.403487  0.518248
3 -0.561381 -0.438716
4  1.002897 -0.452045

In [410]: df.foo1
Out[410]:
0 -0.868315
1 -2.677551
2 -1.403487
3 -0.561381
4  1.002897
Name: foo1, dtype: float64

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

5.2. DataFrame
In [5]: df.fo<TAB>
df.foo1 df.foo2

5.3 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:

5.3.1 From 3D ndarray with optional axis labels

In [411]: 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 [412]: wp
Out[412]:
<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

5.3.2 From dict of DataFrame objects

In [413]: data = {'Item1' : DataFrame(randn(4, 3)),
.....:     'Item2' : DataFrame(randn(4, 2))}

In [414]: Panel(data)
Out[414]:
<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:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intersect</td>
<td>False</td>
<td>drops elements whose indices do not align</td>
</tr>
<tr>
<td>orient</td>
<td>items</td>
<td>use minor to use DataFrames' columns as panel items</td>
</tr>
</tbody>
</table>

For example, compare to the construction above:

```
In [415]: Panel.from_dict(data, orient='minor')
Out[415]:
<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 [416]: df = DataFrame({'a': ['foo', 'bar', 'baz'],
                     'b': np.random.randn(3))
 ....:
In [417]: df
Out[417]:
a b
0 foo 1.448717
1 bar 0.608653
2 baz -1.409338
In [418]: data = {'item1': df, 'item2': df}
In [419]: panel = Panel.from_dict(data, orient='minor')
```

```
In [420]: panel['a']
Out[420]:
     item1  item2
0      foo    foo
1      bar    bar
2      baz    baz
```

```
In [421]: panel['b']
Out[421]:
     item1  item2
0  1.448717  1.448717
1  0.608653  0.608653
2 -1.409338 -1.409338
```

```
In [422]: panel['b'].dtypes
Out[422]:
     item1  item2
dtype: object
```

**Note:** Unfortunately 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. This will get worked on, of course, in future releases. And faster if you join me in working on the codebase.
5.3.3 From DataFrame using to_panel method

This method was introduced in v0.7 to replace LongPanel.to_long, and converts a DataFrame with a two-level index to a Panel.

In [423]: midx = MultiIndex(levels=[['one', 'two'], ['x', 'y']], labels=[[1,1,0,0],[1,0,1,0]])
In [424]: df = DataFrame({'A' : [1, 2, 3, 4], 'B': [5, 6, 7, 8]}, index=midx)
In [425]: df.to_panel()
Out[425]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 2 (minor_axis)
Items axis: A to B
Major_axis axis: one to two
Minor_axis axis: x to y

5.3.4 Item selection / addition / deletion

Similar to DataFrame functioning as a dict of Series, Panel is like a dict of DataFrames:

In [426]: wp['Item1']
Out[426]:
   A  B   C  D
2000-01-01 -1.362139 -0.098512 -0.491067 0.048491
2000-01-02  2.287810 -0.403876 -1.076283 -0.155956
2000-01-03  0.388741 -1.284588 -0.508030 0.841173
2000-01-04 -0.555843 -0.030913 -0.289758 1.318467
2000-01-05  1.025903  0.195796  0.030198 -0.349406

In [427]: 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.

5.3.5 Transposing

A Panel can be rearranged using its transpose method (which does not make a copy by default unless the data are heterogeneous):

In [428]: wp.transpose(2, 0, 1)
Out[428]:
<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

5.3.6 Indexing / Selection

<table>
<thead>
<tr>
<th>Operation</th>
<th>Syntax</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Select item</td>
<td>wp[item]</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at major_axis label</td>
<td>wp.major_xs(val)</td>
<td>DataFrame</td>
</tr>
<tr>
<td>Get slice at minor_axis label</td>
<td>wp.minor_xs(val)</td>
<td>DataFrame</td>
</tr>
</tbody>
</table>
For example, using the earlier example data, we could do:

```
In [429]: wp['Item1']
Out[429]:
    A       B       C       D
2000-01-01 -1.362139 -0.098512 -0.491067  0.048491
2000-01-02  2.287810 -0.403876 -1.076283 -0.155956
2000-01-03  0.388741 -1.284588 -0.508030  0.841173
2000-01-04 -0.555843 -0.030913 -0.289758  1.318467
2000-01-05  1.025903  0.195796  0.030198 -0.349406
```

```
In [430]: wp.major_xs(wp.major_axis[2])
Out[430]:
      Item1     Item2     Item3
A   0.388741  1.076202  0.361216
B -1.284588 -0.464905  2.763121
C -0.508030  0.432658 -1.174206
D  0.841173 -0.623043 -1.350105
```

```
In [431]: wp.minor_axis
Out[431]: Index(['A', 'B', 'C', 'D'], dtype=object)
```

```
In [432]: wp.minor_xs('C')
Out[432]:
      Item1     Item2     Item3
2000-01-01 -0.491067 -0.530157  0.926267
2000-01-02 -1.076283 -0.692498  1.554205
2000-01-03 -0.508030  0.432658 -1.174206
2000-01-04 -0.289758  1.771692 -0.163549
2000-01-05  0.030198 -0.016490 -1.831272
```

### 5.3.7 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 [433]: panel = Panel(np.random.randn(3, 5, 4), items=['one', 'two', 'three'],
                      major_axis=date_range('1/1/2000', periods=5),
                      minor_axis=['a', 'b', 'c', 'd'])
```

```
In [434]: panel.to_frame()
```

```
          one    two    three
major minor
2000-01-01 a  1.219834 -0.842503  1.130688
   b -0.185793 -0.585949  1.348831
   c -1.016665 -0.864916 -0.709279
   d  0.170971  0.031573 -0.125291
2000-01-02 a  0.411316 -1.170645 -1.746865
   b -0.773663 -0.655575 -0.802833
   c -0.028610 -1.297237 -0.824150
   d  0.532592 -1.739996 -0.056603
2000-01-03 a  0.579638 -0.093661  1.443225
   b -1.514892  0.873783 -1.013384
   c  1.528058 -1.803206 -0.591932
   d  0.954347 -0.134374  0.679775
2000-01-04 a -0.635819 -0.100241 -0.921819
```

### 5.3. Panel
5.4 Panel4D (Experimental)

Panel4D is a 4-Dimensional named container very much like a Panel, but having 4 named dimensions. It is intended as a test bed for more N-Dimensional named containers.

- **labels**: axis 0, each item corresponds to a Panel contained inside
- **items**: axis 1, each item corresponds to a DataFrame contained inside
- **major_axis**: axis 2, it is the index (rows) of each of the DataFrames
- **minor_axis**: axis 3, it is the columns of each of the DataFrames

Panel4D is a sub-class of Panel, so most methods that work on Panels are applicable to Panel4D. The following methods are disabled:

- join, to_frame, to_excel, to_sparse, groupby

Construction of Panel4D works in a very similar manner to a Panel

5.4.1 From 4D ndarray with optional axis labels

```
In [435]: 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 [436]: p4d
Out[436]:
<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
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A to D
```

5.4.2 From dict of Panel objects

```
In [437]: data = { 'Label1' : Panel({ 'Item1' : DataFrame(randn(4, 3)) } ),
.....:          'Label2' : Panel({ 'Item2' : DataFrame(randn(4, 2)) } ) }
```

```
In [438]: Panel4D(data)
Out[438]:
<class 'pandas.core.panelnd.Panel4D'>
```
5.4.3 Slicing

Slicing works in a similar manner to a Panel. [] slices the first dimension. .ix allows you to slice arbitrarily and get back lower dimensional objects.

```python
In [439]: p4d['Label1']
Out[439]:
<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
```

4D -> Panel

```python
In [440]: p4d.ix[:,:,:,'A']
Out[440]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 2 (major_axis) x 5 (minor_axis)
Items axis: Label1 to Label2
Major_axis axis: Item1 to Item2
Minor_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```

4D -> DataFrame

```python
In [441]: p4d.ix[:,:,0,'A']
Out[441]:
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Item1</td>
<td>2.301716</td>
</tr>
<tr>
<td>Item2</td>
<td>1.454477</td>
</tr>
</tbody>
</table>
```

4D -> Series

```python
In [442]: p4d.ix[:,0,0,'A']
Out[442]:
Label1 2.301716
Label2 -0.045494
Name: A, dtype: float64
```

5.4.4 Transposing

A Panel4D can be rearranged using its transpose method (which does not make a copy by default unless the data are heterogeneous).

```python
In [443]: p4d.transpose(3, 2, 1, 0)
Out[443]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 4 (labels) x 5 (items) x 2 (major_axis) x 2 (minor_axis)
```
5.5 PanelND (Experimental)

PanelND is a module with a set of factory functions to enable a user to construct N-dimensional named containers like Panel4D, with a custom set of axis labels. Thus a domain-specific container can easily be created.

The following creates a Panel5D. A new panel type object must be sliceable into a lower dimensional object. Here we slice to a Panel4D.

In [444]: from pandas.core import panelnd

In [445]: Panel5D = panelnd.create_nd_panel_factory(
.....:    klass_name = 'Panel5D',
.....:    axis_orders = [ 'cool', 'labels', 'items', 'major_axis', 'minor_axis' ],
.....:    axis_slices = { 'labels' : 'labels', 'items' : 'items',
.....:                   'major_axis' : 'major_axis', 'minor_axis' : 'minor_axis' },
.....:    slicer = Panel4D,
.....:    axis_aliases = { 'major' : 'major_axis', 'minor' : 'minor_axis' },
.....:    stat_axis = 2)

In [446]: p5d = Panel5D(dict(C1 = p4d))

In [447]: p5d
Out[447]:
<class 'pandas.core.panelnd.Panel5D'>
Dimensions: 1 (cool) x 2 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Cool axis: C1 to C1
Labels axis: Label1 to Label2
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

# print a slice of our 5D
In [448]: p5d.ix['C1',:,:,0:3,:]
Out[448]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 2 (labels) x 2 (items) x 3 (major_axis) x 4 (minor_axis)
Labels axis: Label1 to Label2
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

# transpose it
In [449]: p5d.transpose(1,2,3,4,0)
Out[449]:
<class 'pandas.core.panelnd.Panel5D'>
Dimensions: 2 (cool) x 2 (labels) x 5 (items) x 4 (major_axis) x 1 (minor_axis)
Cool axis: Label1 to Label2
Labels axis: Label1 to Label2
Items axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Major_axis axis: A to D
Minor_axis axis: Cl to Cl

# look at the shape & dim
In [450]: p5d.shape
Out[450]: [1, 2, 2, 5, 4]

In [451]: p5d.ndim
Out[451]: 5
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]:**

```python
index = date_range('1/1/2000', periods=8)
```

**In [2]:**

```python
s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
```

**In [3]:**

```python
df = DataFrame(randn(8, 3), index=index, columns=['A', 'B', 'C'])
```

**In [4]:**

```python
wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
           major_axis=date_range('1/1/2000', periods=5),
           minor_axis=['A', 'B', 'C', 'D'])
```

### 6.1 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]:**

```python
long_series = Series(randn(1000))
```

**In [6]:**

```python
long_series.head()
```

```
   0    1.162813
   1    0.870161
   2    2.792723
   3    0.776395
   4   -1.181190
   dtype: float64
```

**In [7]:**

```python
long_series.tail(3)
```

```
  997    0.545698
  998    0.008194
  999   1.452061
  dtype: float64
```
6.2 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 -1.0078   0.56199  0.560802
2000-01-02  0.770819 -0.972052  0.896288
```

```
In [9]: df.columns = [x.lower() for x in df.columns]
In [10]: df
Out[10]:
          a         b         c
2000-01-01 -1.0078   0.56199  0.560802
2000-01-02  0.770819 -0.972052  0.896288
2000-01-03  0.718942  0.329576  1.169925
2000-01-04  0.958928 -0.935316 -0.827036
2000-01-05 -1.240656  0.834546 -0.160635
2000-01-06 -3.590370 -1.247926 -1.445820
2000-01-07 -0.042194  0.906744 -0.471145
2000-01-08 -0.256360 -0.098316 -0.770393
```

To get the actual data inside a data structure, one need only access the *values* property:

```
In [11]: s.values
Out[11]: array([-0.5347, -0.0236, -0.9306, -0.2505, -0.1546])
```

```
In [12]: df.values
Out[12]:
array([[-1.0078, 0.56199, 0.560802],
       [ 0.7708, -0.972052, 0.896288],
       [-0.718942, 0.329576, 1.169925],
       [ 0.958928, -0.935316, -0.827036],
       [-1.240656, 0.834546, -0.160635],
       [-3.590370, -1.247926, -1.445820],
       [-0.042194, 0.906744, -0.471145],
       [-0.256360, -0.098316, -0.770393]])
```

```
In [13]: wp.values
Out[13]:
array([[-0.5695, 0.917 , 0.4495, -0.8452],
       [-0.5009, 0.4569, 0.4477, 0.2638],
       [ 1.3112, -0.0522, 0.508 , -0.7318],
       [-2.1767, -0.5234, -0.2092, -0.1431],
       [-1.0446, 0.5449, 0.0648, 0.4873]],
       [[ 0.0002, -1.3767, -0.7805, 0.6007],
       [-0.8252, 0.4755, 0.7108, -1.3615]])
```
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.

### 6.3 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.

#### 6.3.1 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]: d = {'one' : Series(randn(3), index=['a', 'b', 'c']),
   'two' : Series(randn(4), index=['a', 'b', 'c', 'd']),
   'three' : Series(randn(3), index=['b', 'c', 'd'])
   }

In [15]: df = DataFrame(d)
```

```
In [16]: df
Out[16]:
     one     three     two
a  0.133865  NaN    -0.352795
b -0.319644 -1.325203  0.934622
c  1.083374  0.512254 -1.658054
d    NaN    -0.019298  1.929479
```

```
In [17]: row = df.ix[1]

In [18]: column = df['two']

In [19]: df.sub(row, axis='columns')
```

```
Out[19]:
     one     three     two
a  0.453509  NaN    -1.287417
b  0.000000  0.000000  0.000000
c  1.403018  1.837457 -2.592677
d  1.305906  0.994857  0.000000
```
In [20]: df.sub(row, axis=1)
Out[20]:

    one  three  two
a  0.453509  NaN -1.287417
b  0.000000  0.000000  0.000000
c  1.403018  1.837457 -2.592677
d  NaN  1.305906  0.994857

In [21]: df.sub(column, axis='index')
Out[21]:

    one  three  two
a  0.486660  NaN   0
b -1.254266 -2.259826  0
c  2.741429  2.170308  0
d  NaN -1.948777  0

In [22]: df.sub(column, axis=0)
Out[22]:

    one  three  two
a  0.486660  NaN   0
b -1.254266 -2.259826  0
c  2.741429  2.170308  0
d  NaN -1.948777  0

With Panel, describing the matching behavior is a bit more difficult, so the arithmetic methods instead (and perhaps confusingly?) give you the option to specify the broadcast axis. For example, suppose we wished to demean the data over a particular axis. This can be accomplished by taking the mean over an axis and broadcasting over the same axis:

In [23]: major_mean = wp.mean(axis='major')

In [24]: major_mean
Out[24]:

    Item1  Item2
A -0.596094  0.141658
B  0.268630 -0.239671
C  0.252154 -0.650685
D -0.193792 -0.327499

In [25]: wp.sub(major_mean, axis='major')
Out[25]:
<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

And similarly for axis="items" and axis="minor".

Note: I could be convinced to make the axis argument in the DataFrame methods match the broadcasting behavior of Panel. Though it would require a transition period so users can change their code...

6.3.2 Missing data / operations with fill values

In Series and DataFrame (though not yet in Panel), 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 [26]: df
Out[26]:
   one  three   two
  a  0.133865   NaN -0.352795
  b -0.319644   -1.325203  0.934622
  c  1.083374   0.512254 -1.658054
  d   NaN      0.019298  1.929479

In [27]: df2
Out[27]:
   one  three   two
  a  0.133865  1.000000 -0.352795
  b -0.319644   -1.325203  0.934622
  c  1.083374   0.512254 -1.658054
  d   NaN      0.019298  1.929479

In [28]: df + df2
Out[28]:
   one  three   two
  a  0.267730    NaN -0.705590
  b -0.639288  -2.650407  1.869244
  c  2.166748   1.024507 -3.316109
  d   NaN       -0.038595  3.858958

In [29]: df.add(df2, fill_value=0)
Out[29]:
   one  three   two
  a  0.267730  1.000000 -0.705590
  b -0.639288  -2.650407  1.869244
  c  2.166748   1.024507 -3.316109
  d   NaN     -0.038595  3.858958

6.3.3 Flexible Comparisons

Starting in v0.8, pandas introduced binary comparison methods `eq, ne, lt, gt, le, and ge` to Series and DataFrame whose behavior is analogous to the binary arithmetic operations described above:

In [30]: df.gt(df2)
Out[30]:
   one  three   two
  a   False    False    False
  b   False    False    False
  c   False    False    False
  d   False    False    False

In [31]: df2.ne(df)
Out[31]:
   one  three   two
  a   False     True    False
  b   False    False    False
  c   False    False    False
  d    True     False    False

6.3. Flexible binary operations
6.3.4 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 [32]: df1 = DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
                      'B' : [np.nan, 2., 3., np.nan, 6.]})
In [33]: df2 = DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
                      'B' : [np.nan, np.nan, 3., 4., 6., 8.]})
In [34]: df1
Out[34]:
   A    B
0  1.0 NaN
1  NaN 2.0
2  3.0 3.0
3  5.0 NaN
4  NaN 6.0
In [35]: df2
Out[35]:
   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 [36]: df1.combine_first(df2)
Out[36]:
   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
```

6.3.5 General DataFrame Combine

The `combine_first` method above calls the more general DataFrame method `combine`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (ie, columns whose names are the same).

So, for instance, to reproduce `combine_first` as above:

```
In [37]: combiner = lambda x, y: np.where(isnull(x), y, x)
In [38]: df1.combine(df2, combiner)
```
Out[38]:
    A  B
 0  1  NaN
 1  2   2
 2  3   3
 3  5   4
 4  3   6
 5  7   8

6.4 Descriptive statistics

A large number of methods for computing descriptive statistics and other related operations on \textit{Series}, \textit{DataFrame}, and \textit{Panel}. Most of these are aggregations (hence producing a lower-dimensional result) like \texttt{sum}, \texttt{mean}, and \texttt{quantile}, but some of them, like \texttt{cumsum} and \texttt{cumprod}, produce an object of the same size. Generally speaking, these methods take an \texttt{axis} argument, just like \texttt{ndarray.\{sum, std, ...\}}, but the axis can be specified by name or integer:

- \textbf{Series}: no axis argument needed
- \textbf{DataFrame}: “index” (axis=0, default), “columns” (axis=1)
- \textbf{Panel}: “items” (axis=0), “major” (axis=1, default), “minor” (axis=2)

For example:

\begin{verbatim}
In [39]: df
Out[39]:
    one  three  two
   a  0.133865 NaN -0.352795
   b -0.319644 -1.325203  0.934622
   c  1.083374  0.512254 -1.658054
   d  NaN     -0.019298  1.929479

In [40]: df.mean(0)
Out[40]:
   one    0.299198
   three  -0.277416
   two    0.213313
dtype: float64

In [41]: df.mean(1)
Out[41]:
   a   -0.109465
   b   -0.236742
   c   -0.020809
   d    0.955091
dtype: float64
\end{verbatim}

All such methods have a \texttt{skipna} option signaling whether to exclude missing data (\texttt{True} by default):

\begin{verbatim}
In [42]: df.sum(0, skipna=False)
Out[42]:
    one  NaN
    three NaN
    two   0.853252
dtype: float64

In [43]: df.sum(\texttt{axis}=1, \texttt{skipna}=\texttt{True})
Out[43]:
\end{verbatim}
Combined with the broadcasting / arithmetic behavior, one can describe various statistical procedures, like standardization (rendering data zero mean and standard deviation 1), very concisely:

```python
In [44]: ts_stand = (df - df.mean()) / df.std()
```

```python
In [45]: ts_stand.std()
Out[45]:
   one  threethreetwo
da 1.9102     0.06242      -0.218930
b -0.185779  -0.710225  -0.417895
```

```python
In [46]: xs_stand = df.sub(df.mean(1), axis=0).div(df.std(1), axis=0)
```

```python
In [47]: xs_stand.std(1)
Out[47]:
   a   b   c   d
one 1.0 1.0 1.0 1.0
three 1.0 1.0 1.0 1.0
```

Note that methods like `cumsum` and `cumprod` preserve the location of NA values:

```python
In [48]: df.cumsum()
Out[48]:
   one  three  two
   a  0.13387  NaN -0.352795
   b -0.185779 -1.325203  0.581827
   c  0.897595 -0.812950 -1.076228
   d  NaN  -0.832247  0.853252
```

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>Number of non-null observations</td>
</tr>
<tr>
<td>sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>mad</td>
<td>Mean absolute deviation</td>
</tr>
<tr>
<td>median</td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td>min</td>
<td>Minimum</td>
</tr>
<tr>
<td>max</td>
<td>Maximum</td>
</tr>
<tr>
<td>abs</td>
<td>Absolute Value</td>
</tr>
<tr>
<td>prod</td>
<td>Product of values</td>
</tr>
<tr>
<td>std</td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td>var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>skew</td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td>kurt</td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td>quantile</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>cumsum</td>
<td>Cumulative sum</td>
</tr>
<tr>
<td>cumprod</td>
<td>Cumulative product</td>
</tr>
<tr>
<td>cummax</td>
<td>Cumulative maximum</td>
</tr>
<tr>
<td>cummin</td>
<td>Cumulative minimum</td>
</tr>
</tbody>
</table>

Note that by chance some NumPy methods, like `mean`, `std`, and `sum`, will exclude NAs on Series input by default:

```python
In [49]: np.mean(df['one'])
Out[49]: 0.2991983434218195
```

```python
In [50]: np.mean(df['one'].values)
Out[50]: nan
```

Series also has a method `nunique` which will return the number of unique non-null values:

```python
In [51]: series = Series(randn(500))
In [52]: series[20:500] = np.nan
In [53]: series[10:20] = 5
In [54]: series.nunique()
Out[54]: 11
```

#### 6.4.1 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):

```python
In [55]: series = Series(randn(1000))
In [56]: series[::2] = np.nan
In [57]: series.describe()
Out[57]:
count 500.000000
mean -0.007877
std 0.911618
min -2.400248
25% -0.659261
50% 0.023054
```

### 6.4. Descriptive statistics 95
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

75% 0.610466
max 2.548590
dtype: float64

In [58]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])

In [59]: frame.ix[:, 5] = np.nan

In [60]: frame.describe()

Out[60]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>500.000000</td>
<td>500.000000</td>
<td>500.000000</td>
<td>500.000000</td>
<td>500.000000</td>
</tr>
<tr>
<td>mean</td>
<td>0.027404</td>
<td>-0.062202</td>
<td>-0.085482</td>
<td>0.047872</td>
<td>0.040049</td>
</tr>
<tr>
<td>std</td>
<td>1.009556</td>
<td>0.934708</td>
<td>1.020247</td>
<td>0.997085</td>
<td>0.981213</td>
</tr>
<tr>
<td>min</td>
<td>-2.820839</td>
<td>-2.629643</td>
<td>-2.907401</td>
<td>-2.678674</td>
<td>-2.439790</td>
</tr>
<tr>
<td>25%</td>
<td>-0.699926</td>
<td>-0.660646</td>
<td>-0.746925</td>
<td>-0.646927</td>
<td>-0.580899</td>
</tr>
<tr>
<td>50%</td>
<td>0.037665</td>
<td>-0.062781</td>
<td>-0.029457</td>
<td>-0.020508</td>
<td>0.016222</td>
</tr>
<tr>
<td>75%</td>
<td>0.708078</td>
<td>0.533449</td>
<td>0.571614</td>
<td>0.769260</td>
<td>0.731781</td>
</tr>
<tr>
<td>max</td>
<td>3.169764</td>
<td>2.790953</td>
<td>3.218046</td>
<td>2.766216</td>
<td>2.978102</td>
</tr>
</tbody>
</table>

For a non-numerical Series object, `describe` will give a simple summary of the number of unique values and most frequently occurring values:

In [61]: s = Series(['a', 'a', 'b', 'b', 'a', 'a', np.nan, 'c', 'd', 'a'])

In [62]: s.describe()

Out[62]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>unique</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>top</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>freq</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dtype</td>
<td>object</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

There also is a utility function, `value_range` which takes a DataFrame and returns a series with the minimum/maximum values in the DataFrame.

### 6.4.2 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 [63]: s1 = Series(randn(5))

In [64]: s1

Out[64]:

<table>
<thead>
<tr>
<th>0</th>
<th>0.190816</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.570470</td>
</tr>
<tr>
<td>2</td>
<td>0.579992</td>
</tr>
<tr>
<td>3</td>
<td>-0.570663</td>
</tr>
<tr>
<td>4</td>
<td>0.653770</td>
</tr>
<tr>
<td>dtype: float64</td>
<td></td>
</tr>
</tbody>
</table>

In [65]: s1.idxmin(), s1.idxmax()

Out[65]: (3, 1)

In [66]: df1 = DataFrame(randn(5, 3), columns=['A', 'B', 'C'])

96 Chapter 6. Essential basic functionality
In [67]: df1
Out[67]:
   A     B     C
0 0.010475 -1.886886  0.703759
1  0.567838   0.954075  0.283241
2  0.156650 -1.192535 -1.015856
3  0.413254 -0.530874  0.030274
4 -0.298383 -0.866317 -0.725995

In [68]: df1.idxmin(axis=0)
Out[68]:
A  4
B  0
C  2
dtype: int64

In [69]: df1.idxmax(axis=1)
Out[69]:
  0  C
  1  B
  2  A
  3  A
  4  A
dtype: object

When there are multiple rows (or columns) matching the minimum or maximum value, `idxmin` and `idxmax` return the first matching index:

In [70]: df3 = DataFrame([2, 1, 1, 3, np.nan], columns=['A'], index=list('edcba'))

In [71]: df3
Out[71]:
   A
e 2
d 1
c 1
b 3
a NaN

In [72]: df3['A'].idxmin()
Out[72]: 'd'

6.4.3 Value counts (histogramming)

The `value_counts` Series method and top-level function computes a histogram of a 1D array of values. It can also be used as a function on regular arrays:

In [73]: data = np.random.randint(0, 7, size=50)

In [74]: data
Out[74]:
array([2, 3, 1, 4, 0, 4, 0, 2, 3, 0, 3, 3, 4, 0, 3, 6, 6, 0, 6, 2, 0, 0,
       0, 6, 2, 0, 2, 4, 2, 3, 0, 6, 5, 1, 6, 3, 6, 6, 4, 2, 3, 1, 6, 5, 5,
       2, 0, 4, 5])

In [75]: s = Series(data)
In [76]: s.value_counts()
Out[76]:
0 11
6 9
3 9
2 8
4 6
5 4
1 3
dtype: int64

In [77]: value_counts(data)
Out[77]:
0 11
6 9
3 9
2 8
4 6
5 4
1 3
dtype: int64

6.4.4 Discretization and quantiling

Continuous values can be discretized using the `cut` (bins based on values) and `qcut` (bins based on sample quantiles) functions:

In [78]: arr = np.random.randn(20)

In [79]: factor = cut(arr, 4)

In [80]: factor
Out[80]: Categorical:
array([(-1.841, -0.823], (-0.823, 0.19], (0.19, 1.204], (1.204, 2.218],
(-1.841, -0.823], (-0.823, 0.19], (0.19, 1.204],
(0.19, 1.204], (0.19, 1.204], (-0.823, 0.19], (-1.841, -0.823],
(-0.823, 0.19], (0.19, 1.204], (1.204, 2.218],
(1.204, 2.218], (-0.823, 0.19], (-1.841, -0.823], (-0.823, 0.19]], dtype=object)
Levels (4): Index([(-1.841, -0.823], (-0.823, 0.19], (0.19, 1.204],
(1.204, 2.218]], dtype=object)

In [81]: factor = cut(arr, [-5, -1, 0, 1, 5])

In [82]: factor
Out[82]: Categorical:
array([(-5, -1], (-1, 0], (0, 1], (1, 5], [-5, -1], (-1, 0],
(0, 1], (1, 5], (0, 1], (-1, 0], (-5, -1], (0, 1]], dtype=object)
Levels (4): Index([(-5, -1], (-1, 0], (0, 1], (1, 5]], dtype=object)

qcut computes sample quantiles. For example, we could slice up some normally distributed data into equal-size quartiles like so:

In [83]: arr = np.random.randn(30)
In [84]: factor = qcut(arr, [0, .25, .5, .75, 1])

In [85]: factor
Out[85]:
Categorical:
array([-2.506, -0.591, -0.145, 0.333, 1.453], dtype=object)

In [86]: value_counts(factor)
Out[86]:
(0.333, 1.453] 8
[-2.506, -0.591] 8
(-0.145, 0.333] 7
(-0.591, -0.145] 7
dtype: int64

6.5 Function application

Arbitrary functions can be applied along the axes of a DataFrame or Panel using the `apply` method, which, like the descriptive statistics methods, take an optional `axis` argument:

In [87]: df.apply(np.mean)
Out[87]:
one 0.299198
three -0.277416
two 0.213313
dtype: float64

In [88]: df.apply(np.mean, axis=1)
Out[88]:
a -0.109465
b -0.236742
c -0.020809
d 0.955091
dtype: float64

In [89]: df.apply(lambda x: x.max() - x.min())
Out[89]:
one 1.403018
three 1.837457
two 3.587534
dtype: float64

In [90]: df.apply(np.cumsum)
Out[90]:
one three two
a 0.133865 NaN -0.352795
b -0.185779 -1.325203 0.581827
In [91]: df.apply(np.exp)
Out[91]:
      one   three   two
a  1.14324  NaN   0.70272
b  0.72641  0.26689  2.56251
c  2.95463  1.66905  0.19051
d  NaN   0.98089  6.88592

Depending on the return type of the function passed to `apply`, the result will either be of lower dimension or the same dimension.
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 [92]: tsdf = DataFrame(randn(1000, 3), columns=['A', 'B', 'C'],
                      index=date_range('1/1/2000', periods=1000))

In [93]: tsdf.apply(
                  lambda x: x.index[x.dropna().argmax()])
Out[93]:
   A         B         C
2000-11-22 00:00:00 2001-09-03 00:00:00 2002-05-01 00:00:00
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:

```python
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 [94]: tsdf
Out[94]:
      A         B         C
2000-01-01  1.162731  0.246389 -0.834775
2000-01-02  1.434571  1.158517  1.031740
2000-01-03 -0.187711 -0.206570  1.435722
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  1.378860 -1.534015  0.464984
2000-01-09  0.496435 -0.344982 -0.178994
2000-01-10  0.369649 -0.345704 -1.047580

In [95]: tsdf.apply(Series.interpolate)
Out[95]:
      A         B         C
2000-01-01  1.162731  0.246389 -0.834775
2000-01-02  1.434571  1.158517  1.031740
2000-01-03 -0.187711 -0.206570  1.435722
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.

See Also:
The section on GroupBy demonstrates related, flexible functionality for grouping by some criterion, applying, and combining the results into a Series, DataFrame, etc.

6.5.1 Applying elementwise Python 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 [96]: f = lambda x: len(str(x))

In [97]: df['one'].map(f)
Out[97]:
   a  14
   b  15
   c  13
   d   3
Name: one, dtype: int64

In [98]: df.applymap(f)
Out[98]:
     one three two
   a  14  3  15
   b  15 14  13
   c  13 14  14
   d   3 16  13
```

Series.map has an additional feature which is that it can be used to easily “link” or “map” values defined by a secondary series. This is closely related to merging/joining functionality:

```
In [99]: s = Series(['six', 'seven', 'six', 'seven', 'six'], index=['a', 'b', 'c', 'd', 'e'])
In [100]: t = Series({'six' : 6., 'seven' : 7.})

In [101]: s
Out[101]:
   a  six
   b  seven
   c  six
   d  seven
   e  six
dtype: object
```
6.6 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:

```python
In [103]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [104]: s
Out[104]:
    a  1.143520
    b  0.143515
    c  1.717025
    d -0.366994
    e -1.255767
    dtype: float64

In [105]: s.reindex(['e', 'b', 'f', 'd'])
Out[105]:
    e  -1.255767
    b  0.143515
    f      NaN
    d  -0.366994
    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:

```python
In [106]: df
Out[106]:
   one         three         two
   a  0.133865  NaN          -0.352795
   b -0.319644  1.325203       0.934622
   c  1.083374  0.512254      -1.658054
   d  NaN    -0.019298     1.929479

In [107]: df.reindex(index=['c', 'f', 'b'], columns=['three', 'two', 'one'])
Out[107]:
   three         two         one
   c  1.083374   0.512254  -1.658054
   f       NaN         NaN     1.929479
   b -0.319644  1.325203   -0.352795
```
For convenience, you may utilize the `reindex_axis` method, which takes the labels and a keyword `axis` parameter.

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:

```python
In [108]: rs = s.reindex(df.index)

In [109]: rs
Out[109]:
    a  1.143520
    b  0.143515
    c  1.717025
dtype: float64

In [110]: rs.index is df.index
Out[110]: True
```

This means that the reindexed Series’s index is the same Python object as the DataFrame’s index.

See Also:

`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.

6.6.1 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:

```python
In [111]: df
Out[111]:
   one  three  two
  a  0.133865   NaN -0.352795
  b -0.319644 -1.325203  0.934622
  c  1.083374  0.512254 -1.658054
d  NaN     -0.019298  1.929479

In [112]: df2
Out[112]:
     one   two
   a -0.165333  0.005947
   b -0.618842  1.293365
   c  0.784176 -1.299312

In [113]: df.reindex_like(df2)
Out[113]:
```
### 6.6.2 Reindexing with `reindex_axis`

### 6.6.3 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
- `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 [114]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [115]: s1 = s[:4]
In [116]: s2 = s[1:]

In [117]: s1.align(s2)
Out[117]:
(a 0.615848
 b -0.016043
 c -1.447277
 d 0.946345
dtype: float64,
    a NaN
    b -0.016043
    c -1.447277
    d 0.946345
dtype: float64)

In [118]: s1.align(s2, join='inner')
Out[118]:
(b -0.016043
c -1.447277
d 0.946345
dtype: float64,
    b -0.016043
c -1.447277
d 0.946345
dtype: float64)

In [119]: s1.align(s2, join='left')
Out[119]:
(a 0.615848
 b -0.016043
 c 1.083374
dtype: float64,
    a 0.615848
    b -0.016043
    c 1.083374
    d NaN
dtype: float64)
```

---

### One two

<table>
<thead>
<tr>
<th>one</th>
<th>two</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 1.083374</td>
<td>-1.658054</td>
</tr>
<tr>
<td>b -0.319644</td>
<td>0.934622</td>
</tr>
<tr>
<td>c -0.352795</td>
<td>0.133865</td>
</tr>
</tbody>
</table>
c  -1.447277
d  0.946345
dtype: float64,
a    NaN
b  -0.016043
c  -1.447277
d  0.946345
dtype: float64)

For DataFrames, the join method will be applied to both the index and the columns by default:

In [120]: df.align(df2, join='inner')
Out[120]:
  ( one  two
   a  0.133865 -0.352795
   b -0.319644  0.934622
   c  1.083374 -1.658054,  
      one  two
     a -0.165333  0.005947
     b -0.618842  1.293365
     c  0.784176 -1.299312)

You can also pass an axis option to only align on the specified axis:

In [121]: df.align(df2, join='inner', axis=0)
Out[121]:
  ( one  three  two
   a  0.133865  NaN  -0.352795
   b -0.319644 -1.325203  0.934622
   c  1.083374  0.512254 -1.658054,  
      one  two
     a -0.165333  0.005947
     b -0.618842  1.293365
     c  0.784176 -1.299312
     d  NaN  -0.019298  1.929479,

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 [122]: df.align(df2.ix[0], axis=1)
Out[122]:
  ( one  three  two
   a  0.133865  NaN  -0.352795
   b -0.319644 -1.325203  0.934622
   c  1.083374  0.512254 -1.658054
   d  NaN  -0.019298  1.929479,
     one  NaN
     two  0.005947
Name: a, dtype: float64)

### 6.6.4 Filling while reindexing

`reindex` takes an optional parameter `method` which is a filling method chosen from the following table:

<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad / ffill</td>
<td>Fill values forward</td>
</tr>
<tr>
<td>bfill / backfill</td>
<td>Fill values backward</td>
</tr>
</tbody>
</table>

6.6. Reindexing and altering labels
Other fill methods could be added, of course, but these are the two most commonly used for time series data. In a way they only make sense for time series or otherwise ordered data, but you may have an application on non-time series data where this sort of “interpolation” logic is the correct thing to do. More sophisticated interpolation of missing values would be an obvious extension.

We illustrate these fill methods on a simple TimeSeries:

```
In [123]: rng = date_range('1/3/2000', periods=8)

In [124]: ts = Series(randn(8), index=rng)

In [125]: ts2 = ts[[0,3,6]]

In [126]: ts
Out[126]:
2000-01-03    0.990340
2000-01-04    -0.070005
2000-01-05    -0.157860
2000-01-06     0.233077
2000-01-07     0.475897
2000-01-08    -1.029480
2000-01-09    -1.079405
2000-01-10    -0.079334
Freq: D, dtype: float64
```

```
In [127]: ts2
Out[127]:
2000-01-03    0.990340
2000-01-06     0.233077
2000-01-09    -1.079405
dtype: float64
```

```
In [128]: ts2.reindex(ts.index)
Out[128]:
2000-01-03    0.990340
2000-01-04     NaN
2000-01-05     NaN
2000-01-06     0.233077
2000-01-07     NaN
2000-01-08     NaN
2000-01-09    -1.079405
2000-01-10     NaN
Freq: D, dtype: float64
```

```
In [129]: ts2.reindex(ts.index, method='ffill')
Out[129]:
2000-01-03    0.990340
2000-01-04    0.990340
2000-01-05    0.990340
2000-01-06    0.233077
2000-01-07    0.233077
2000-01-08    0.233077
2000-01-09    -1.079405
2000-01-10    -1.079405
Freq: D, dtype: float64
```

```
In [130]: ts2.reindex(ts.index, method='bfill')
Out[130]:
2000-01-03    0.990340
2000-01-04    0.990340
2000-01-05    0.990340
2000-01-06    0.233077
2000-01-07    0.233077
2000-01-08    0.233077
2000-01-09    -1.079405
2000-01-10    -1.079405
Freq: D, dtype: float64
```
Note the same result could have been achieved using `fillna`:

```
In [131]: ts2.reindex(ts.index).fillna(method='ffill')
Out[131]:
2000-01-03  0.990340
2000-01-04  0.990340
2000-01-05  0.990340
2000-01-06  0.233077
2000-01-07  0.233077
2000-01-08  0.233077
2000-01-09 -1.079405
2000-01-10 -1.079405
Freq: D, dtype: float64
```

Note these methods generally assume that the indexes are sorted. They may be modified in the future to be a bit more flexible but as time series data is ordered most of the time anyway, this has not been a major priority.

### 6.6.5 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 [132]: df
Out[132]:
  one  three  two
  a  0.133865 NaN -0.352795
  b  -0.319644  1.325203  0.934622
  c   1.083374  0.512254 -1.658054
  d   NaN       -0.019298  1.929479

In [133]: df.drop(['a', 'd'], axis=0)
Out[133]:
  one  three  two
  b  -0.319644 -1.325203  0.934622
  c   1.083374  0.512254 -1.658054

In [134]: df.drop(['one'], axis=1)
Out[134]:
    three  two
  a  NaN -0.352795
  b -1.325203  0.934622
  c  0.512254 -1.658054
  d -0.019298  1.929479
```

Note that the following also works, but is a bit less obvious / clean:

```
In [135]: df.reindex(df.index - ['a', 'd'])
Out[135]:
  one  three  two
  a  NaN -0.352795
  b  NaN -0.352795
  c  NaN -0.352795
  d  NaN -0.352795
```

### 6.6. Reindexing and altering labels
6.6.6 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 [136]: s
Out[136]:
     a   0.615848
     b  -0.016043
     c  -1.447277
     d   0.946345
     e   0.723322
dtype: float64

In [137]: s.rename(str.upper)
Out[137]:
     A   0.615848
     B  -0.016043
     C  -1.447277
     D   0.946345
     E   0.723322
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). But if you pass a dict or Series, it need only contain a subset of the labels as keys:

In [138]: df.rename(columns={'one' : 'foo', 'two' : 'bar'},
                  index={'a' : 'apple', 'b' : 'banana', 'd' : 'durian'})
Out[138]:
        foo   three    bar
apple  0.133865  NaN  -0.352795
banana -0.319644 -1.325203  0.934622
c      1.083374  0.512254 -1.658054
durian NaN   -0.019298  1.929479

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. The Panel class has a related rename_axis class which can rename any of its three axes.

6.7 Iteration

Because Series is array-like, basic iteration produces the values. Other data structures follow the dict-like convention of iterating over the “keys” of the objects. In short:

- **Series**: values
- **DataFrame**: column labels
- **Panel**: item labels

Thus, for example:
6.7.1 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 [140]: for item, frame in wp.iteritems():
    ....:    print item
    ....:    print frame

Item1
 A    B    C   D
2000-01-01 -0.569502 0.916952 0.449538 -0.845226
2000-01-02 -0.500946 0.456865 0.447653 0.263834
2000-01-03  1.311241 -0.052172 0.508033 -0.731786
2000-01-04 -2.176710 -0.523424 -0.209228 -0.143088
2000-01-05 -1.044551  0.544929  0.064773  0.487304

Item2
 A    B    C   D
2000-01-01  0.000200 -1.376720 -0.780456  0.600739
2000-01-02 -0.825227  0.475548  0.710782 -1.361472
2000-01-03 -0.419611 -0.700988 -1.304530 -0.253342
2000-01-04  0.074107 -1.384211 -0.587086 -0.356223
2000-01-05  1.878822  1.788014 -1.292132 -0.267198
```
For instance, a contrived way to transpose the dataframe would be:

```python
In [142]: df2 = DataFrame({'x': [1, 2, 3], 'y': [4, 5, 6]})
```

```python
In [143]: print df2
   x  y
0 1  4
1 2  5
2 3  6
```

```python
In [144]: print df2.T
   0 1 2
   x 1 2 3
   y 4 5 6
```

```python
In [145]: df2_t = DataFrame(dict((idx, values) for idx, values in df2.iterrows()))
```

```python
In [146]: print df2_t
   0 1 2
   x 1 2 3
   y 4 5 6
```

### 6.7.3 `itertuples`

This method will return an iterator yielding a tuple 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 proper.

For instance,

```python
In [147]: for r in df2.itertuples(): print r
 (0, 1, 4)
 (1, 2, 5)
 (2, 3, 6)
```

### 6.8 Vectorized string methods

Series is equipped (as of pandas 0.8.1) 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) build-in string methods:

```python
In [148]: s = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])
```

```python
In [149]: s.str.lower()
Out[149]:
0    a
1    b
2    c
3   aaba
4    baca
5   NaN
6   caba
```
In [150]: s.str.upper()
Out[150]:
0  A
1  B
2  C
3  AABA
4  BACA
5  NaN
6  CABA
7  DOG
8  CAT
dtype: object

In [151]: s.str.len()
Out[151]:
  0  1
  1  1
  2  1
  3  4
  4  4
  5  NaN
  6  4
  7  3
  8  3
dtype: float64

Methods like split return a Series of lists:

In [152]: s2 = Series(['a_b_c', 'c_d_e', np.nan, 'f_g_h'])

In [153]: s2.str.split('_')
Out[153]:
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 [154]: s2.str.split('_').str.get(1)
Out[154]:
0  b
1  d
2  NaN
3  g
dtype: object

In [155]: s2.str.split('_').str[1]
Out[155]:
0  b
1  d
2  NaN
3  g
dtype: object

6.8. Vectorized string methods
Methods like `replace` and `findall` take regular expressions, too:

```python
In [156]: s3 = Series(['A', 'B', 'C', 'Aaba', 'Baca', 
            ....:       '', np.nan, 'CABA', 'dog', 'cat'])

In [157]: s3
Out[157]:
0     A
1     B
2     C
3    Aaba
4    Baca
5      
6   NaN
7   CABA
8     dog
9     cat
dtype: object
```

```python
In [158]: s3.str.replace('^a|dog', 'XX-XX ', case=False)
Out[158]:
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
```

Methods like `contains`, `startswith`, and `endswith` takes an extra `na` argument so missing values can be considered True or False:

```python
In [159]: s4 = Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', 'dog', 'cat'])

In [160]: s4.str.contains('A', na=False)
Out[160]:
0    True
1   False
2   False
3    True
4   False
5   False
6    True
7   False
8   False
dtype: bool
```
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>Concatenate strings</td>
</tr>
<tr>
<td>split</td>
<td>Split strings on delimiter</td>
</tr>
<tr>
<td>get</td>
<td>Index into each element (retrieve i-th element)</td>
</tr>
<tr>
<td>join</td>
<td>Join strings in each element of the Series with passed separator</td>
</tr>
<tr>
<td>contains</td>
<td>Return boolean array if each string contains pattern/regex</td>
</tr>
<tr>
<td>replace</td>
<td>Replace occurrences of pattern/regex with some other string</td>
</tr>
<tr>
<td>repeat</td>
<td>Duplicate values (s.str.repeat(3) equivalent to x * 3)</td>
</tr>
<tr>
<td>pad</td>
<td>Add whitespace to left, right, or both sides of strings</td>
</tr>
<tr>
<td>center</td>
<td>Equivalent to pad(side='both')</td>
</tr>
<tr>
<td>slice</td>
<td>Slice each string in the Series</td>
</tr>
<tr>
<td>slice_replace</td>
<td>Replace slice in each string with passed value</td>
</tr>
<tr>
<td>count</td>
<td>Count occurrences of pattern</td>
</tr>
<tr>
<td>startswith</td>
<td>Equivalent to str.startswith(pat) for each element</td>
</tr>
<tr>
<td>endswith</td>
<td>Equivalent to str.endswith(pat) for each element</td>
</tr>
<tr>
<td>findall</td>
<td>Compute list of all occurrences of pattern/regex for each string</td>
</tr>
<tr>
<td>match</td>
<td>Call re.match on each element, returning matched groups as list</td>
</tr>
<tr>
<td>len</td>
<td>Compute string lengths</td>
</tr>
<tr>
<td>strip</td>
<td>Equivalent to str.strip</td>
</tr>
<tr>
<td>rstrip</td>
<td>Equivalent to str.rstrip</td>
</tr>
<tr>
<td>lstrip</td>
<td>Equivalent to str.lstrip</td>
</tr>
<tr>
<td>lower</td>
<td>Equivalent to str.lower</td>
</tr>
<tr>
<td>upper</td>
<td>Equivalent to str.upper</td>
</tr>
</tbody>
</table>

### 6.9 Sorting by index and value

There are two obvious kinds of sorting that you may be interested in: sorting by label and sorting by actual values. The primary method for sorting axis labels (indexes) across data structures is the `sort_index` method.

```python
In [161]: unsorted_df = df.reindex(index=['a', 'd', 'c', 'b'],
                               columns=['three', 'two', 'one'])

In [162]: unsorted_df.sort_index()
Out[162]:
          three  two     one
     a    NaN -0.352795  0.133865
     b -1.325203  0.934622 -0.319644
     c  0.512254 -1.658054  1.083374
     d -0.019298  1.929479    NaN

In [163]: unsorted_df.sort_index(ascending=False)
Out[163]:
          three  two     one
     d -0.019298  1.929479    NaN
     c  0.512254 -1.658054  1.083374
     b -1.325203  0.934622 -0.319644
     a    NaN -0.352795  0.133865

In [164]: unsorted_df.sort_index(axis=1)
Out[164]:
          one  three  two
     a  0.133865    NaN -0.352795
     d    NaN  0.019298  1.929479
```

### 6.9. Sorting by index and value
DataFrame.sort_index can accept an optional by argument for axis=0 which will use an arbitrary vector or a column name of the DataFrame to determine the sort order:

```
In [165]: df.sort_index(by='two')
Out[165]:
   one  three  two
  c  1.083374  0.512254 -1.658054
  a  0.133865  NaN -0.352795
  b -0.319644 -1.325203  0.934622
  d  NaN -0.019298  1.929479
```

The by argument can take a list of column names, e.g.:

```
In [166]: df = DataFrame({'one':[2,1,1,1],'two':[1,3,2,4],'three':[5,4,3,2]})
In [167]: df[['one', 'two', 'three']].sort_index(by=['one','two'])
Out[167]:
   one  two  three
  2   1    2    3
  1   1    3    4
  3   1    4    2
  0   2    1    5
```

Series has the method order (analogous to R’s order function) which sorts by value, with special treatment of NA values via the na_last argument:

```
In [168]: s[2] = np.nan
In [169]: s.order()
Out[169]:
   0    A
   3   Aaba
   1    B
   4   Baca
   6   CABA
   8    cat
   7    dog
  2   NaN
  5   NaN
dtype: object
In [170]: s.order(na_last=False)
Out[170]:
   2   NaN
   5   NaN
   0    A
   3   Aaba
   1    B
   4   Baca
   6   CABA
   8    cat
   7    dog
dtype: object
```

Some other sorting notes / nuances:

- Series.sort sorts a Series by value in-place. This is to provide compatibility with NumPy methods which
expect the \texttt{ndarray.sort} behavior.

- \texttt{DataFrame.sort} takes a column argument instead of \texttt{by}. This method will likely be deprecated in a future release in favor of just using \texttt{sort_index}.

## 6.10 Copying, type casting

The \texttt{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 \texttt{values} attribute or advanced indexing

To be clear, no pandas methods have the side effect of modifying your data; almost all methods return new objects, leaving the original object untouched. If data is modified, it is because you did so explicitly.

Data can be explicitly cast to a NumPy dtype by using the \texttt{astype} method or alternately passing the \texttt{dtype} keyword argument to the object constructor.

```python
In [171]: df = DataFrame(np.arange(12).reshape((4, 3)))
In [172]: df[0].dtype
Out[172]: dtype('int64')
In [173]: df.astype(float)[0].dtype
Out[173]: dtype('float64')
In [174]: df = DataFrame(np.arange(12).reshape((4, 3)), dtype=float)
In [175]: df[0].dtype
Out[175]: dtype('float64')
```

### 6.10.1 Inferring better types for object columns

The \texttt{convert_objects} DataFrame method will attempt to convert \texttt{dtype=object} columns to a better NumPy dtype. Occasionally (after transposing multiple times, for example), a mixed-type DataFrame will end up with everything as \texttt{dtype=object}. This method attempts to fix that:

```python
In [176]: df = DataFrame(randn(6, 3), columns=['a', 'b', 'c'])
```

```python
In [177]: df['d'] = 'foo'
```

```python
In [178]: df
Out[178]:
   a    b     c    d
0  1.031643 -0.189461 -0.437520   foo
1  0.239650  0.056665 -0.950583   foo
2  0.406598 -1.327319 -0.764997   foo
3  0.619450 -0.158757  1.182297   foo
4  0.345184  0.096056  0.724360   foo
5 -2.790083 -0.168660  0.039725   foo
```

```python
In [179]: df = df.T.T
```

6.10. Copying, type casting
In [180]: df.dtypes
Out[180]:
   a    object
   b    object
   c    object
   d    object
dtype: object

In [181]: converted = df.convert_objects()

In [182]: converted.dtypes
Out[182]:
   a    float64
   b    float64
   c    float64
   d    object
dtype: object

6.11 Pickling and serialization

All pandas objects are equipped with save methods which use Python’s cPickle module to save data structures to disk using the pickle format.

In [183]: df
Out[183]:
   a      b         c          d
0  1.031643 -0.1894613 -0.4375196  foo
1  0.2396501  0.05666547 -0.950583  foo
2  0.4065984 -1.327319  -0.7649967  foo
3  0.6194499 -0.1587574  1.182297  foo
4  0.345184  0.09605619  0.7243603  foo
5 -2.790083 -0.1686605  0.03972503 foo

In [184]: df.save('foo.pickle')

The load function in the pandas namespace can be used to load any pickled pandas object (or any other pickled object) from file:

In [185]: load('foo.pickle')
Out[185]:
   a      b         c          d
0  1.031643 -0.1894613 -0.4375196  foo
1  0.2396501  0.05666547 -0.950583  foo
2  0.4065984 -1.327319  -0.7649967  foo
3  0.6194499 -0.1587574  1.182297  foo
4  0.345184  0.09605619  0.7243603  foo
5 -2.790083 -0.1686605  0.03972503 foo

There is also a save function which takes any object as its first argument:

In [186]: save(df, 'foo.pickle')

In [187]: load('foo.pickle')
Out[187]:
   a      b         c          d
0  1.031643 -0.1894613 -0.4375196  foo
6.12 Working with package options

Introduced in 0.10.0, pandas supports a new system for working with options. 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:

```python
In [188]: import pandas as pd
In [189]: pd.options.display.max_rows
Out[189]: 100
In [190]: pd.options.display.max_rows = 999
In [191]: pd.options.display.max_rows
Out[191]: 999
```

There is also an API composed of 4 relevant functions, available directly from the `pandas` namespace, and they are:

- `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.

**Note:** developers can check out pandas/core/config.py for more info.

but all of the functions above accept a regexp pattern (`re.search` style) as argument, so passing in a substring will work - as long as it is unambiguous:

```python
In [192]: get_option("display.max_rows")
Out[192]: 999
In [193]: set_option("display.max_rows",101)
In [194]: get_option("display.max_rows")
Out[194]: 101
In [195]: set_option("max_r",102)
In [196]: get_option("display.max_rows")
Out[196]: 102
```

However, the following will **not work** because it matches multiple option names, e.g. `'display.max_colwidth'`, `display.max_rows`, `display.max_columns`:

```python
In [197]: try:
.....:     get_option("display.max_")
.....:     except KeyError as e:
.....:         print(e)
.....: File "<ipython-input-197-7cbb78c48d28>", line 3
except KeyError as e:
```
The docstrings of all the functions document the available options, but you can also get a list of available options and their descriptions with `describe_option`. When called with no argument `describe_option` will print out descriptions for all available options.

In [198]: describe_option()

display.chop_threshold: [default: None] [currently: None]
 : float or None
   if set to a float value, all float values smaller than the given threshold
   will be displayed as exactly 0 by repr and friends.
display.colheader_justify: [default: right] [currently: right]
 : 'left'/'right'
   Controls the justification of column headers. used by DataFrameFormatter.
display.column_space: [default: 12] [currently: 12]
 : No description available.
display.date_dayfirst: [default: False] [currently: False]
 : boolean
   When True, prints and parses dates with the day first, eg 20/01/2005
display.date_yearfirst: [default: False] [currently: False]
 : boolean
   When True, prints and parses dates with the year first, eg 2005/01/20
display.encoding: [default: UTF-8] [currently: UTF-8]
 : 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.
display.expand_frame_repr: [default: True] [currently: True]
 : boolean
   Whether to print out the full DataFrame repr for wide DataFrames
   across multiple lines.
   If False, the summary representation is shown.
display.float_format: [default: None] [currently: None]
 : 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 core.format.EngFormatter for an example.
display.line_width: [default: 80] [currently: 80]
 : int
   When printing wide DataFrames, this is the width of each line.
display.max_columns: [default: 20] [currently: 20]
 : int
   max_rows and max_columns are used in __repr__() methods to decide if
to_string() or info() is used to render an object to a string.
   Either one, or both can be set to 0 (experimental). Pandas will figure
   out how big the terminal is and will not display more rows or/and
   columns that can fit on it.
display.max_colwidth: [default: 50] [currently: 50]
 : 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.
display.max_info_columns: [default: 100] [currently: 100]
 : int
   max_info_columns is used in DataFrame.info method to decide if
per column information will be printed.

display.max_info_rows: [default: 1000000] [currently: 1000000]
: int or None

max_info_rows is the maximum number of rows for which a frame will
perform a null check on its columns when repr’ing To a console.
The default is 1,000,000 rows. So, if a DataFrame has more
1,000,000 rows there will be no null check performed on the
columns and thus the representation will take much less time to
display in an interactive session. A value of None means always
perform a null check when repr’ing.

display.max_rows: [default: 100] [currently: 102]
: int

This sets the maximum number of rows pandas should output when printing
out various output. For example, this value determines whether the repr()
for a dataframe prints out fully or just an summary repr.

display.max_seq_items: [default: None] [currently: None]
: int or None

when pretty-printing a long sequence, no more then '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.

display.multi_sparse: [default: True] [currently: True]
: boolean

"sparsify" MultiIndex display (don’t display repeated
elements in outer levels within groups)

display.notebook_repr_html: [default: True] [currently: True]
: boolean

When True, IPython notebook will use html representation for
pandas objects (if it is available).

display.pprint_nest_depth: [default: 3] [currently: 3]
: int

Controls the number of nested levels to process when pretty-printing

display.precision: [default: 7] [currently: 7]
: int

Floating point output precision (number of significant digits). This is
only a suggestion

mode.sim_interactive: [default: False] [currently: False]
: boolean

Whether to simulate interactive mode for purposes of testing

mode.use_inf_as_null: [default: False] [currently: False]
: boolean

True means treat None, NaN, INF, -INF as null (old way),
False means None and NaN are null, but INF, -INF are not null
(new way).

or you can get the description for just the options that match the regexp you pass in:

In [199]: describe_option("date")

display.date_dayfirst: [default: False] [currently: False]
: boolean

When True, prints and parses dates with the day first, eg 20/01/2005

display.date_yearfirst: [default: False] [currently: False]
: boolean

When True, prints and parses dates with the year first, eg 2005/01/20

All options also have a default value, and you can use the reset_option to do just that:
In [200]: get_option("display.max_rows")
Out[200]: 100

In [201]: set_option("display.max_rows", 999)

In [202]: get_option("display.max_rows")
Out[202]: 999

In [203]: reset_option("display.max_rows")

In [204]: get_option("display.max_rows")
Out[204]: 100

and you also set multiple options at once:

In [205]: reset_option("^display\.")

6.13 Console Output Formatting

Note: set_printoptions/ reset_printoptions are now deprecated (but functioning), and both, as well as set_eng_float_format, use the options API behind the scenes. The corresponding options now live under “print.XYZ”, and you can set them directly with get/set_option.

Use the set_eng_float_format function in the pandas.core.common module to alter the floating-point formatting of pandas objects to produce a particular format.

For instance:

In [206]: set_eng_float_format(accuracy=3, use_eng_prefix=True)

In [207]: df['a']/1.e3
Out[207]:
0  1.032m
1  239.650u
2  406.598u
3  619.450u
4  345.184u
5 -2.790m
Name: a, dtype: object

In [208]: df['a']/1.e6
Out[208]:
0  1.032u
1  239.650n
2  406.598n
3  619.450n
4  345.184n
5 -2.790u
Name: a, dtype: object

The set_printoptions function has a number of options for controlling how floating point numbers are formatted (using the precision argument) in the console and . The max_rows and max_columns control how many rows and columns of DataFrame objects are shown by default. If max_columns is set to 0 (the default, in fact), the library will attempt to fit the DataFrame’s string representation into the current terminal width, and defaulting to the summary view otherwise.
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 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 / chapter, 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. Expect more work to be invested higher-dimensional data structures (including Panel) in the future, especially in label-based advanced indexing.

7.1 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. Thus,

- **Series**: series[label] returns a scalar value
- **DataFrame**: frame[colname] returns a Series corresponding to the passed column name
- **Panel**: panel[itemname] returns a DataFrame corresponding to the passed item name

Here we construct a simple time series data set to use for illustrating the indexing functionality:

```python
In [620]: dates = np.asarray(date_range('1/1/2000', periods=8))

In [621]: df = DataFrame(randn(8, 4), index=dates, columns=['A', 'B', 'C', 'D'])

In [622]: df
Out[622]:
          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 -1.344312
2000-01-08 -0.370647  -1.157892 -1.344312  0.844885
```
In [623]: panel = Panel({'one': df, 'two': df - df.mean()})

In [624]: panel
Out[624]:
<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 [625]: s = df['A']

In [626]: s[dates[5]]
Out[626]: -0.67368970808837059

In [627]: panel['two']
Out[627]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.409571</td>
<td>0.113086</td>
<td>-0.610826</td>
<td>-0.936507</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1.152571</td>
<td>0.222735</td>
<td>1.017442</td>
<td>-0.845111</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.921390</td>
<td>-1.708620</td>
<td>0.403304</td>
<td>1.270929</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.662014</td>
<td>-0.310822</td>
<td>-0.141342</td>
<td>0.470985</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.484513</td>
<td>0.962970</td>
<td>1.174465</td>
<td>-0.888276</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.733231</td>
<td>0.509598</td>
<td>-0.580194</td>
<td>0.724113</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.345164</td>
<td>0.972995</td>
<td>-0.816769</td>
<td>-0.840143</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-0.430188</td>
<td>-0.761943</td>
<td>-0.446079</td>
<td>1.044010</td>
</tr>
</tbody>
</table>

7.1.1 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 get_value method, which is implemented on all of the data structures:

In [628]: s.get_value(dates[5])
Out[628]: -0.67368970808837059

In [629]: df.get_value(dates[5], 'A')
Out[629]: -0.67368970808837059

There is an analogous set_value method which has the additional capability of enlarging an object. This method always returns a reference to the object it modified, which in the case of enlargement, will be a new object:

In [630]: df.set_value(dates[5], 'E', 7)
Out[630]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
<td>-1.135632</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>1.121212</td>
<td>-0.173215</td>
<td>0.119209</td>
<td>-1.044236</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
<td>1.071804</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
<td>0.271860</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
<td>-1.087401</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.673690</td>
<td>0.113648</td>
<td>-1.478427</td>
<td>0.524988</td>
<td>7</td>
</tr>
</tbody>
</table>
7.1.2 Additional Column Access

You may access a column on a dataframe directly as an attribute:

```
In [631]: df.A
Out[631]:
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
2000-01-06    0.673690
2000-01-07    0.404705
2000-01-08   -0.370647
Name: A, dtype: float64
```

If you are using the IPython environment, you may also use tab-completion to see the accessible columns of a DataFrame.

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 [632]: df[['B', 'A']] = df[['A', 'B']]
```

You may find this useful for applying a transform (in-place) to a subset of the columns.

7.1.3 Data slices on other axes

It’s certainly possible to retrieve data slices along the other axes of a DataFrame or Panel. We tend to refer to these slices as cross-sections. DataFrame has the `xs` function for retrieving rows as Series and Panel has the analogous
**7.1.4 Slicing ranges**

The most robust and consistent way of slicing ranges along arbitrary axes is described in the `Advanced indexing` section detailing the `.ix` 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 [639]: s[:5]
Out[639]:
2000-01-01 -0.282863
2000-01-02 -0.173215
2000-01-03 -2.104569
2000-01-04 -0.706771
2000-01-05  0.567020
Name: A, dtype: float64
```

```
In [640]: s[:2]
Out[640]:
2000-01-01 -0.282863
2000-01-03 -2.104569
2000-01-05  0.567020
2000-01-07  0.577046
Name: A, dtype: float64
```

```
In [641]: s[::-1]
```

```
Out[641]:
2000-01-08  -1.157892
2000-01-07   0.577046
2000-01-06   0.113648
2000-01-05   0.567020
2000-01-04  -0.706771
2000-01-03  -2.104569
2000-01-02  -0.173215
2000-01-01  -0.282863
Name: A, dtype: float64

Note that setting works as well:

In [642]: s2 = s.copy()

In [643]: s2[:5] = 0

In [644]: s2

Out[644]:
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  0.113648
2000-01-07  0.577046
2000-01-08  -1.157892
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 [645]: df[:3]

Out[645]:
     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

In [646]: df[::-1]

Out[646]:
     A     B     C     D
2000-01-08 -1.157892 -0.370647 -1.344312 0.844885
2000-01-07  0.577046 0.404705 -1.715002 -1.039268
2000-01-06  0.113648 -0.673690 -1.478427 0.524988
2000-01-05  0.567020 -0.424972  0.276232 -1.087401
2000-01-04  0.706771  0.721555 -1.039575 0.271860
2000-01-03 -2.104569 -0.861849 -0.494929 1.071804
2000-01-02 -0.173215 1.212112  0.119209 -1.044236
2000-01-01 -0.282863 0.469112 -1.509059 -1.135632

7.1.5 Boolean indexing

Another common operation is the use of boolean vectors to filter the data.

Using a boolean vector to index a Series works exactly as in a numpy ndarray:
```
In [647]: s[s > 0]
Out[647]:
2000-01-05  0.567020
2000-01-06  0.113648
2000-01-07  0.577046
Name: A, dtype: float64

In [648]: s[(s < 0) & (s > -0.5)]
Out[648]:
2000-01-01 -0.282863
2000-01-02 -0.173215
Name: A, dtype: float64

In [649]: s[(s < -1) | (s > 1 )]
Out[649]:
2000-01-03 -2.104569
2000-01-08 -1.157892
Name: A, dtype: float64

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 [650]: df[df['A'] > 0]
Out[650]:
     A       B       C       D
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

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 [651]: criterion = df2['a'].map(lambda x: x.startswith('t'))
In [652]: df2[criterion]
```
```
In [651]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
.....: 'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
.....: 'c' : randn(7)})
.....:
In [652]: df2['a'].isin(['one', 'two'])
```
```
List comprehensions and `map` method of Series can also be used to produce more complex criteria:
```
# only want 'two' or 'three'
In [653]: criterion = df2['a'].map(lambda x: x.startswith('t'))
```
```
In [654]: df2[criterion]
```
```
# equivalent but slower
```
In [655]: df2[[x.startswith('t') for x in df2['a']]]
Out[655]:
   a  b  c
2  two  y  1.643563
3  three  x -1.469388
4  two  y  0.357021

# Multiple criteria
In [656]: df2[criterion & (df2['b'] == 'x')]
Out[656]:
   a  b  c
3  three  x -1.469388

Note, with the `advanced indexing` `ix` method, you may select along more than one axis using boolean vectors combined with other indexing expressions.

### 7.1.6 Where 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`.

# return only the selected rows
In [657]: s[s > 0]
Out[657]:
      2000-01-05  0.567020
      2000-01-06  0.113648
      2000-01-07  0.577046
Name: A, dtype: float64

# return a Series of the same shape as the original
In [658]: s.where(s > 0)
Out[658]:
      2000-01-01 NaN
      2000-01-02 NaN
      2000-01-03 NaN
      2000-01-04 NaN
      2000-01-05  0.567020
      2000-01-06  0.113648
      2000-01-07  0.577046
      2000-01-08 NaN
Name: A, 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.

# return a DataFrame of the same shape as the original
# this is equivalent to `'df.where(df < 0)`'
In [659]: df[df < 0]
Out[659]:
     A  B    C    D
2000-01-01 -0.282863 NaN -1.509059 -1.135632
2000-01-02 -0.173215 NaN NaN -1.044236
2000-01-03 -2.104569 -0.861849 -0.494929 NaN
2000-01-04 -0.706771 NaN -1.039575 NaN
2000-01-05 NaN -0.424972 NaN -1.087401
2000-01-06 NaN -0.673690 -1.478427 NaN
2000-01-07 NaN NaN -1.715002 -1.039268

### 7.1. Basics

127
In addition, `where` takes an optional `other` argument for replacement of values where the condition is False, in the returned copy.

```python
In [660]: df.where(df < 0, -df)
Out[660]:
   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 wish to set values based on some boolean criteria. This can be done intuitively like so:

```python
In [661]: s2 = s.copy()

In [662]: s2[s2 < 0] = 0

In [663]: s2
Out[663]:
2000-01-01 0.000000
2000-01-02 0.000000
2000-01-03 0.000000
2000-01-04 0.000000
2000-01-05 0.567020
2000-01-06 0.113648
2000-01-07 0.577046
2000-01-08 0.000000
Name: A, dtype: float64

In [664]: df2 = df.copy()

In [665]: df2[df2 < 0] = 0

In [666]: df2
Out[666]:
   A        B        C        D
2000-01-01 0.000000 0.469112 0.000000 0.000000
2000-01-02 0.000000 1.212112 0.119209 0.000000
2000-01-03 0.000000 0.000000 0.000000 1.071804
2000-01-04 0.000000 0.721555 0.000000 0.271860
2000-01-05 0.567020 0.000000 0.276232 0.000000
2000-01-06 0.113648 0.000000 0.524988 0.000000
2000-01-07 0.577046 0.404705 0.000000 0.000000
2000-01-08 0.000000 0.000000 0.844885

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 `.ix` (but on the contents rather than the axis labels).

```python
In [667]: df2 = df.copy()

In [668]: df2[df2[1:4] > 0] = 3

In [669]: df2
```
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 [670]: df_orig = df.copy()

In [671]: df_orig.where(df > 0, -df, inplace=True);
In [671]: df_orig
```

```
Out[671]:
       A         B         C         D
2000-01-01 -0.282863  0.469112 -1.509059 -1.135632
2000-01-02 -0.173215  3.000000  3.000000  3.000000
2000-01-03 -2.104569 -0.861849 -0.494929  3.000000
2000-01-04 -0.706771  3.000000 -1.039575  3.000000
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 -0.706771  0.404705 -1.344312  0.844885
```

`mask` is the inverse boolean operation of `where`.

```
In [672]: s.mask(s >= 0)
```

```
Out[672]:
2000-01-01  -0.282863
2000-01-02  -0.173215
2000-01-03  -2.104569
2000-01-04  -0.706771
2000-01-05  NaN
2000-01-06  NaN
2000-01-07  NaN
2000-01-08  -1.157892
Name: A, dtype: float64
```

```
In [673]: df.mask(df >= 0)
```

```
Out[673]:
       A         B         C         D
2000-01-01   NaN      -1.509059   NaN   -1.135632
2000-01-02   NaN      -1.044236   NaN   -1.044236
2000-01-03  -0.861849   NaN   -1.478427   NaN
2000-01-04  -0.706771   NaN   -1.344312   NaN
2000-01-05   NaN    -1.039268   NaN
2000-01-06   NaN   -1.087401   NaN
2000-01-07   NaN   -1.039268   NaN
2000-01-08   NaN   -1.344312   NaN
```

7.1. Basics 129
7.1.7 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.

```python
In [674]: index = Index(randint(0, 1000, 10))

In [675]: index
Out[675]: Int64Index([969, 412, 496, 195, 288, 101, 881, 900, 732, 658], dtype=int64)

In [676]: positions = [0, 9, 3]

In [677]: index[positions]
Out[677]: Int64Index([969, 658, 195], dtype=int64)

In [678]: index.take(positions)
Out[678]: Int64Index([969, 658, 195], dtype=int64)

In [679]: ser = Series(randn(10))

In [680]: ser.ix[positions]
Out[680]:
0 -0.968914
9 -1.131345
3  1.247642
dtype: float64

In [681]: ser.take(positions)
Out[681]:
0 -0.968914
9 -1.131345
3  1.247642
dtype: float64
```

For DataFrames, the given indices should be a 1d list or ndarray that specifies row or column positions.

```python
In [682]: frm = DataFrame(randn(5, 3))

In [683]: frm.take([1, 4, 3])
Out[683]:
0  1  2
1 -0.932132  1.956030  0.017587
4 -0.077118 -0.408530 -0.862495
3 -1.143704  0.215897  1.193555

In [684]: frm.take([0, 2], axis=1)
Out[684]:
0  2
0 -0.089329 -0.945867
1 -0.932132  0.017587
2 -0.016692  0.254161
3 -1.143704  1.193555
4 -0.077118 -0.862495
```

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 [685]: arr = randn(10)

In [686]: arr.take([False, False, True, True])
Out[686]: array([ 1.3461,  1.3461,  1.5118,  1.5118])

In [687]: arr[[0, 1]]
Out[687]: array([ 1.3461,  1.5118])

In [688]: ser = Series(randn(10))

In [689]: ser.take([False, False, True, True])
Out[689]:
   0  -0.105381
   0  -0.105381
   1  -0.532532
   1  -0.532532
   dtype: float64

In [690]: ser.ix[[0, 1]]
Out[690]:
   0  -0.105381
   1  -0.532532
   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.

7.1.8 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.

duplicated returns a boolean vector whose length is the number of rows, and which indicates whether a row is duplicated.

drop_duplicates removes duplicate rows.

By default, the first observed row of a duplicate set is considered unique, but each method has a take_last parameter that indicates the last observed row should be taken instead.

In [691]: df2 = DataFrame({'a' : ['one', 'one', 'two', 'three', 'two', 'one', 'six'],
                          'b' : ['x', 'y', 'y', 'x', 'y', 'x', 'x'],
                          'c' : np.random.randn(7)})

In [692]: df2.duplicated(['a', 'b'])
Out[692]:
   0    False
   1    False
   2    False
   3    False
   4    True
   5    True
   6    False
   dtype: bool

In [693]: df2.drop_duplicates(['a', 'b'])
In [694]: df2.drop_duplicates([‘a’, ‘b’], take_last=True)
Out[694]:
    a  b  c
0  one x -0.339355
1  one y  0.593616
2  two y  0.884345
3  three x  1.591431
5  one x  0.220390
6  six x  0.435589

7.1.9 Dictionary-like get method

Each of Series, DataFrame, and Panel have a get method which can return a default value.

In [695]: s = Series([1,2,3], index=[‘a’, ‘b’, ‘c’])
In [696]: s.get(‘a’)  # equivalent to s[‘a’]
Out[696]: 1
In [697]: s.get(‘x’, default=-1)
Out[697]: -1

7.2 Advanced indexing with labels

We have avoided excessively overloading the [] / __getitem__ operator to keep the basic functionality of the pandas objects straightforward and simple. However, there are often times when you may wish get a subset (or analogously set a subset) of the data in a way that is not straightforward using the combination of reindex and []. Complicated setting operations are actually quite difficult because reindex usually returns a copy.

By advanced indexing we are referring to a special .ix attribute on pandas objects which enable you to do getting/setting operations on a DataFrame, for example, with matrix/ndarray-like semantics. Thus you can combine the following kinds of indexing:

- An integer or single label, e.g. 5 or ‘a’
- A list or array of labels [‘a’, ‘b’, ‘c’] or integers [4, 3, 0]
- A slice object with ints 1:7 or labels ‘a’:'f'
- A boolean array

We’ll illustrate all of these methods. First, note that this provides a concise way of reindexing on multiple axes at once:

In [698]: subindex = dates[[3,4,5]]
In [699]: df.reindex(index=subindex, columns=[‘C’, ‘B’])
Out[699]:
   C    B
2000-01-04 -1.039575  0.721555
2000-01-05  0.276232 -0.424972
In [700]: df.ix[subindex, ['C', 'B']]
Out[700]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-04</td>
<td>-1.039575</td>
<td>0.721555</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.276232</td>
<td>-0.424972</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-1.478427</td>
<td>-0.673690</td>
</tr>
</tbody>
</table>

Assignment / setting values is possible when using `ix`:

In [701]: df2 = df.copy()

In [702]: df2.ix[subindex, ['C', 'B']] = 0

In [703]: df2
Out[703]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-0.282863</td>
<td>0.469112</td>
<td>-1.509059</td>
<td>-1.135632</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
<td>1.212112</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-2.104569</td>
<td>-0.861849</td>
<td>-0.494929</td>
<td>1.071804</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.271860</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>0.000000</td>
<td>0.000000</td>
<td>-1.087401</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.113648</td>
<td>0.000000</td>
<td>0.000000</td>
<td>0.524988</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.577046</td>
<td>0.404705</td>
<td>-1.715002</td>
<td>-1.039268</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>-1.157892</td>
<td>-0.370647</td>
<td>-1.344312</td>
<td>0.844885</td>
</tr>
</tbody>
</table>

Indexing with an array of integers can also be done:

In [704]: df.ix[[4,3,1]]

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>-0.424972</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
<td>0.721555</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
<td>1.212112</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
</tbody>
</table>
```

In [705]: df.ix[dates[[4,3,1]]]

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>-0.424972</td>
<td>0.276232</td>
<td>-1.087401</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
<td>0.721555</td>
<td>-1.039575</td>
<td>0.271860</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
<td>1.212112</td>
<td>0.119209</td>
<td>-1.044236</td>
</tr>
</tbody>
</table>
```

**Slicing** has standard Python semantics for integer slices:

In [706]: df.ix[1:7, :2]

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-02</td>
<td>-0.173215</td>
<td>1.212112</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>-2.104569</td>
<td>-0.861849</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-0.706771</td>
<td>0.721555</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.567020</td>
<td>-0.424972</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>0.113648</td>
<td>-0.673690</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>0.577046</td>
<td>0.404705</td>
</tr>
</tbody>
</table>
```

Slicing with labels is semantically slightly different because the slice start and stop are **inclusive** in the label-based case:

In [707]: date1, date2 = dates[[2, 4]]
In [708]: print date1, date2
1970-01-11 232:00:00 1970-01-11 24:00:00

In [709]: df.ix[datel:date2]
Out[709]:
Empty DataFrame
Columns: [A, B, C, D]
Index: []

In [710]: df[‘A’].ix[datel:date2]
Out[710]: Series([], dtype: float64)

Getting and setting rows in a DataFrame, especially by their location, is much easier:

In [711]: df2 = df[:5].copy()

In [712]: df2.ix[3]
Out[712]:
A -0.706771
B  0.721555
C -1.039575
D  0.271860
Name: 2000-01-04 00:00:00, dtype: float64

In [713]: df2.ix[3] = np.arange(len(df2.columns))

In [714]: df2
Out[714]:
   A     B     C     D
0 0.00 -1.00 -2.00 -3.00
1 1.00 -1.00 -2.00 -3.00
2 2.00 -1.00 -2.00 -3.00
3 3.00 -1.00 -2.00 -3.00
4 4.00 -1.00 -2.00 -3.00

Column or row selection can be combined as you would expect with arrays of labels or even boolean vectors:

In [715]: df.ix[df[‘A’] > 0, ‘B’]
Out[715]:
2000-01-05 -0.424972
2000-01-06 -0.673690
2000-01-07  0.404705
Name: B, dtype: float64

In [716]: df.ix[datel:date2, ‘B’]
Out[716]: Series([], dtype: float64)

In [717]: df.ix[datel, ‘B’]
Out[717]: -0.861849

Slicing with labels is closely related to the truncate method which does precisely .ix[start:stop] but returns a copy (for legacy reasons).

7.2.1 Returning a view versus a copy

The rules about when a view on the data is returned are entirely dependent on NumPy. Whenever an array of labels or a boolean vector are involved in the indexing operation, the result will be a copy. With single label / scalar indexing and slicing, e.g. df.ix[3:6] or df.ix[‘’, ‘A’], a view will be returned.
7.2.2 The select method

Another way to extract slices from an object is with the `select` method of Series, DataFrame, and Panel. This method should be used only when there is no more direct way. `select` takes a function which operates on labels along `axis` and returns a boolean. For instance:

```
In [718]: df.select(lambda x: x == 'A', axis=1)
Out[718]:
   A
2000-01-01 -0.282863
2000-01-02 -0.173215
2000-01-03 -2.104569
2000-01-04 -0.706771
2000-01-05  0.567020
2000-01-06  0.113648
2000-01-07  0.577046
2000-01-08 -1.157892
```

7.2.3 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 [719]: dflookup = DataFrame(np.random.rand(20,4), columns = ['A','B','C','D'])
In [720]: dflookup.lookup(xrange(0,10,2), ['B','C','A','B','D'])
Out[720]: array([ 0.0227, 0.4199, 0.529 , 0.9674, 0.5357])
```

7.2.4 Advanced indexing with integer labels

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 `.ix`. The following code will generate exceptions:

```
s = Series(range(5))
s[-1]
df = DataFrame(np.random.randn(5, 4))
df
df.ix[-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).

7.2.5 Setting values in mixed-type DataFrame

Setting values on a mixed-type DataFrame or Panel is supported when using scalar values, though setting arbitrary vectors is not yet supported:

```
In [721]: df2 = df[:4]

In [722]: df2['foo'] = 'bar'

In [723]: print df2
```
7.3 Index objects

The pandas Index class and its subclasses can be viewed as implementing an orderd set in addition to providing the support infrastructure necessary for lookups, data alignment, and reindexing. The easiest way to create one directly is to pass a list or other sequence to `Index`:

```
In [727]: index = Index([‘e’, ‘d’, ‘a’, ‘b’])
```

```
In [728]: index
Out[728]: Index([‘e’, ‘d’, ‘a’, ‘b’], dtype=object)
```

```
In [729]: ‘d’ in index
Out[729]: True
```

You can also pass a name to be stored in the index:

```
In [730]: index = Index([‘e’, ‘d’, ‘a’, ‘b’], name=’something’)
```

```
In [731]: index.name
Out[731]: ‘something’
```

Starting with pandas 0.5, the name, if set, will be shown in the console display:

```
In [732]: index = Index(range(5), name=’rows’)
```

```
In [733]: columns = Index([‘A’, ‘B’, ‘C’], name=’cols’)
```

```
In [734]: df = DataFrame(np.random.randn(5, 3), index=index, columns=columns)
```

```
In [735]: df
Out[735]:
   cols  A   B   C
  rows
0  NaN  NaN  NaN
1  NaN  NaN  NaN
2  NaN  NaN  NaN
3  NaN  NaN  NaN
4  NaN  NaN  NaN
```
7.3.1 Set operations on Index objects

The three main operations are union (\(|\)), intersection (\(\&\)), and diff (\(-\)). These can be directly called as instance methods or used via overloaded operators:

```
In [737]: a = Index(['c', 'b', 'a'])

In [738]: b = Index(['c', 'e', 'd'])

In [739]: a.union(b)
Out[739]: Index(['a', 'b', 'c', 'd', 'e'], dtype=object)

In [740]: a | b
Out[740]: Index(['a', 'b', 'c', 'd', 'e'], dtype=object)

In [741]: a & b
Out[741]: Index(['c'], dtype=object)

In [742]: a - b
Out[742]: Index(['a', 'b'], dtype=object)
```

7.3.2 isin method of Index objects

One additional operation is the isin method that works analogously to the Series.isin method found here.

7.4 Hierarchical indexing (MultiIndex)

Hierarchical indexing (also referred to as “multi-level” indexing) is brand new in the pandas 0.4 release. It 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 the 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.
Note: Given that hierarchical indexing is so new to the library, it is definitely “bleeding-edge” functionality but is certainly suitable for production. But, there may inevitably be some minor API changes as more use cases are explored and any weaknesses in the design / implementation are identified. pandas aims to be “eminently usable” so any feedback about new functionality like this is extremely helpful.

7.4.1 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 an array of tuples where each tuple is unique. A MultiIndex can be created from a list of arrays (using MultiIndex.from_arrays) or an array of tuples (using MultiIndex.from_tuples).

```
In [743]: arrays = [['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux'],
            ['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two']]

In [744]: tuples = zip(*arrays)

In [745]: tuples
Out[745]: [('bar', 'one'),
            ('bar', 'two'),
            ('baz', 'one'),
            ('baz', 'two'),
            ('foo', 'one'),
            ('foo', 'two'),
            ('qux', 'one'),
            ('qux', 'two')]

In [746]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [747]: s = Series(randn(8), index=index)

In [748]: s
Out[748]:
first   second
bar  one   -0.223540
      two   0.542054
baz  one  -0.688585
      two  -0.352676
foo  one  -0.711411
      two  -2.122599
qux  one   1.962935
      two   1.672027
dtype: float64
```

As a convenience, you can pass a list of arrays directly into Series or DataFrame to construct a MultiIndex automatically:

```
In [749]: arrays = [np.array(['bar', 'bar', 'baz', 'baz', 'foo', 'foo', 'qux', 'qux']),
            np.array(['one', 'two', 'one', 'two', 'one', 'two', 'one', 'two'])]

In [750]: s = Series(randn(8), index=arrays)

In [751]: s
```
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

Out[751]:
bar    one  -0.880984
two    0.997289
baz    one  -1.693316
two    -0.179129
foo    one  -1.598062
two    0.936914
qux    one   0.912560
two  -1.003401
dtype: float64

In [752]: df = DataFrame(randn(8, 4), index=arrays)

In [753]: df
Out[753]:
       0  1  2
bar one  1.632781 -0.724626 0.178219 0.310610
two -0.108002 -0.974226 -1.147708 -2.281374
baz one  0.760010 -0.742532 1.533318 2.495362
two -0.432771 -0.068954 0.112246 0.102578
foo one  0.871721 -0.816064 -0.784880 1.030659
two  1.187483 -1.939946 0.377312 0.734122
qux one  2.141616 -0.011225 0.048869 -1.360687
two -0.479010 -0.859661 -0.231595 -0.527750

All of the MultiIndex constructors accept a names argument which stores string names for the levels themselves. If no names are provided, some arbitrary ones will be assigned:

In [754]: index.names
Out[754]: ['first', 'second']

This index can back any axis of a pandas object, and the number of levels of the index is up to you:

In [755]: df = DataFrame(randn(3, 8), index=['A', 'B', 'C'], columns=index)

In [756]: df
Out[756]:
       first  bar  baz  foo  qux
second one  1.296337 0.150680 0.123836 0.571764
        two -1.993606 -1.927385 -2.027924 1.624972
      one  0.150680 0.123836 0.571764 -0.823761
      two  0.150680 0.123836 0.571764 0.310610
     one  0.123836 0.571764 -0.823761 0.535420
     two  0.123836 0.571764 -0.823761 -1.032853
    A  0.571764 -0.823761 0.535420 -1.032853
    B -1.993606 -1.927385 -2.027924 1.624972
    C  0.150680 0.123836 0.571764 -0.823761
   one  1.469725 1.304124 1.449735 0.203109
   two -1.032011 0.969818 -0.962723 1.382083
  A  1.304124 1.449735 0.203109 -1.032011
  B  1.469725 1.304124 1.449735 0.203109
  C -1.032011 0.969818 -0.962723 1.382083
  one  0.669142 -0.433567 -0.273610 0.680433
  two  0.669142 -0.433567 -0.273610 -0.308450
  one  1.155563 -0.823761 0.535420 -1.032853
  two  1.155563 -0.823761 0.535420 -1.032853
 first second
bar one  -1.99360 6 -1.927385 -2.027924 1.624972 0.551135 3.059267
      two  0.455264 -0.030740 0.935716 1.061192 -2.107852 0.199905
 baz one  0.323586 -0.641630 -0.587514 0.053897 0.194889 -0.381994
     two  0.318587 2.089075 -0.728293 -0.090255 -0.748199 1.318931
foo one  0.912560 -0.912560 -0.912560 -0.912560 -0.912560 -0.912560
      two  0.095031 -0.270099 -0.707140 -0.738882 0.229453 0.304418

We’ve “sparsified” the higher levels of the indexes to make the console output a bit easier on the eyes.

It's worth keeping in mind that there's nothing preventing you from using tuples as atomic labels on an axis:

7.4. Hierarchical indexing (MultiIndex)
In [758]: Series(randn(8), index=tuples)
Out[758]:
(bar, one) 0.736135
(bar, two) -0.859631
(baz, one) -0.424100
(baz, two) -0.776114
(foo, one) 1.279293
(foo, two) 0.943798
(qux, one) -1.001859
(qux, two) 0.306546
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.

Note that how the index is displayed can be controlled using the multi_sparse option in pandas.set_printoptions:

In [759]: pd.set_printoptions(multi_sparse=False)

In [760]: df
Out[760]:
   first  bar  bar  baz  baz  foo  foo  qux  qux
second
   A -1.296337 0.150680 0.123836 0.571764 1.555563 -0.823761 0.535420 -1.032853
   B  1.469725 1.304124 1.449735 0.203109 -1.032011 0.969818 -0.962723 1.382083
   C -0.938794 0.669142 -0.433567 -0.273610 0.680433 -0.308450 -0.276099 -1.821168

In [761]: pd.set_printoptions(multi_sparse=True)

7.4.2 Reconstructing the level labels

The method get_level_values will return a vector of the labels for each location at a particular level:

In [762]: index.get_level_values(0)
Out[762]: Index([bar, bar, baz, Baz, foo, foo, qux, qux], dtype=object)

In [763]: index.get_level_values('second')
Out[763]: Index([one, two, one, two, one, two, one, two], dtype=object)

7.4.3 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 [764]: df[‘bar’]
Out[764]:
   second  one  two
   A -1.296337 0.150680
   B  1.469725 1.304124
   C -0.938794 0.669142

In [765]: df[‘bar’, ‘one’]
7.4.4 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 [768]: s + s[:-2]
Out[768]:
bar one -1.761968
two  1.994577
baz one -3.386631
two  0.358257
foo one -3.196125
two  1.873828
qux one NaN
two  NaN
dtype: float64
```

```
In [769]: s + s[::2]
Out[769]:
bar one -1.761968
two  NaN
baz one -3.386631
two  NaN
foo one -3.196125
two  NaN
qux one  1.825119
two  NaN
dtype: float64
```

`reindex` can be called with another MultiIndex or even a list or array of tuples:

```
In [770]: s.reindex(index[:3])
Out[770]:
first  second
bar one  -0.880984
        two  0.997289
baz one  -1.693316
dtype: float64
```
7.4.5 Advanced indexing with hierarchical index

Syntactically integrating MultiIndex in advanced indexing with .ix is a bit challenging, but we’ve made every effort to do so. For example the following works as you would expect:

```
In [772]: df = df.T

In [773]: df
Out[773]:
     A     B     C
first second
bar  one   -1.296337 1.469725 -0.938794
two   0.150680 1.304124  0.669142
baz  one   0.123836 1.449735 -0.433567
two   0.571764 0.203109 -0.273610
foo  one   1.555563 -1.032011  0.680433
two  -0.823761 0.969818 -0.308450
qux  one   0.535420 -0.962723 -0.276099
two  -1.032853 1.382083 -1.821168

In [774]: df.ix['bar']
Out[774]:
     A     B     C
second
one   -1.296337 1.469725 -0.938794
two    0.150680 1.304124  0.669142

In [775]: df.ix[('baz', 'two'):('qux', 'one')]
Out[775]:
     A     B     C
first second
baz  one   0.123836 1.449735 -0.433567
two   0.571764 0.203109 -0.273610
foo  one   1.555563 -1.032011  0.680433
two  -0.823761 0.969818 -0.308450
qux  one   0.535420 -0.962723 -0.276099
two  -1.032853 1.382083 -1.821168
```

“Partial” slicing also works quite nicely:

```
In [776]: df.ix['baz':'foo']
Out[776]:
     A     B     C
first second
baz  one   0.123836 1.449735 -0.433567
two   0.571764 0.203109 -0.273610
foo  one   1.555563 -1.032011  0.680433
two  -0.823761 0.969818 -0.308450

In [777]: df.ix[('baz', 'two'):('qux', 'one')]
Out[777]:
     A     B     C
first second
baz  one   0.123836 1.449735 -0.433567
two   0.571764 0.203109 -0.273610
qux  one   0.535420 -0.962723 -0.276099
two  -1.032853 1.382083 -1.821168
```
baz  two  0.571764  0.203109 -0.273610
foo  one  1.555563 -1.032011  0.680433
two  -0.823761  0.969818 -0.308450
qux  one  0.535420  -0.962723 -0.276099

In [778]: df.ix[('baz', 'two'): 'foo']
Out[778]:
   A     B     C
first second
baz  two  0.571764  0.203109 -0.273610
foo  one  1.555563 -1.032011  0.680433
two  -0.823761  0.969818 -0.308450

Passing a list of labels or tuples works similar to reindexing:

In [779]: df.ix[['bar', 'two'], ('qux', 'one')]
Out[779]:
   A     B     C
first second
bar  two  0.15068  1.304124  0.669142
qux  one  0.53542 -0.962723 -0.276099

The following does not work, and it’s not clear if it should or not:

>>> df.ix[['bar', 'qux']]

The code for implementing .ix makes every attempt to “do the right thing” but as you use it you may uncover corner cases or unintuitive behavior. If you do find something like this, do not hesitate to report the issue or ask on the mailing list.

### 7.4.6 Cross-section with hierarchical index

The xs method of DataFrame additionally takes a level argument to make selecting data at a particular level of a MultiIndex easier.

In [780]: df.xs('one', level='second')
Out[780]:
   A     B     C
first
bar -1.296337  1.469725 -0.938794
baz  0.123836  1.449735 -0.433567
foo  1.555563 -1.032011  0.680433
qux  0.535420 -0.962723 -0.276099

### 7.4.7 Advanced reindexing and alignment with hierarchical index

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:

In [781]: midx = MultiIndex(levels=[['zero', 'one'], ['x', 'y']],
                        labels=[[1,1,0,0],[1,0,1,0]])

In [782]: df = DataFrame(randn(4,2), index=midx)

In [783]: print df

7.4. Hierarchical indexing (MultiIndex)
0 1
one y 0.307453 -0.906534
x -1.505397 1.392009
zero y -0.027793 -0.631023
x -0.662357 2.725042

In [784]: df2 = df.mean(level=0)

In [785]: print df2
    0    1
zero -0.345075 1.047010
one -0.598972 0.242737

In [786]: print df2.reindex(df.index, level=0)
    0    1
one y -0.598972 0.242737
x -0.598972 0.242737
zero y -0.345075 1.047010
x -0.345075 1.047010

In [787]: df_aligned, df2_aligned = df.align(df2, level=0)

In [788]: print df_aligned
    0    1
one y 0.307453 -0.906534
x -1.505397 1.392009
zero y -0.027793 -0.631023
x -0.662357 2.725042

In [789]: print df2_aligned
    0    1
one y -0.598972 0.242737
x -0.598972 0.242737
zero y -0.345075 1.047010
x -0.345075 1.047010

7.4.8 The need for sortedness

Caveat emptor: the present implementation of MultiIndex requires that the labels be sorted for some of the slicing / indexing routines to work correctly. You can think about breaking the axis into unique groups, where at the hierarchical level of interest, each distinct group shares a label, but no two have the same label. However, the MultiIndex does not enforce this: you are responsible for ensuring that things are properly sorted. There is an important new method sortlevel to sort an axis within a MultiIndex so that its labels are grouped and sorted by the original ordering of the associated factor at that level. Note that this does not necessarily mean the labels will be sorted lexicographically!

In [790]: import random; random.shuffle(tuples)

In [791]: s = Series(randn(8), index=MultiIndex.from_tuples(tuples))

In [792]: s
Out [792]:
baz one -1.847240
two -0.529247
foo two 0.614656
bar two -1.590742
```
qux one  -0.156479
foo one   -1.696377
qux two   0.819712
bar one   -2.107728
dtype: float64

In [793]: s.sortlevel(0)
Out[793]:
bar one   -2.107728
  two   -1.590742
baz one   -1.847240
  two   -0.529247
foo one   -1.696377
  two    0.614656
qux one   -0.156479
  two    0.819712
dtype: float64

In [794]: s.sortlevel(1)
Out[794]:
bar one   -2.107728
baz one   -1.847240
foo one   -1.696377
qux one   -0.156479
  two   -1.590742
  two   -0.529247
  two    0.614656
  two    0.819712
dtype: float64

Note, you may also pass a level name to sortlevel if the MultiIndex levels are named.

In [795]: s.index.names = ['L1', 'L2']

In [796]: s.sortlevel(level='L1')
Out[796]:
L1   L2
bar one   -2.107728
  two   -1.590742
baz one   -1.847240
  two   -0.529247
foo one   -1.696377
  two    0.614656
qux one   -0.156479
  two    0.819712
dtype: float64

In [797]: s.sortlevel(level='L2')
Out[797]:
L1   L2
bar one   -2.107728
baz one   -1.847240
foo one   -1.696377
qux one   -0.156479
  two   -1.590742
  two   -0.529247
  two    0.614656
  two    0.819712
```
Some indexing will work even if the data are not sorted, but will be rather inefficient and will also return a copy of the data rather than a view:

```python
In [798]: s['qux']
Out[798]:
L2
one -0.156479
two 0.819712
dtype: float64
```

```python
In [799]: s.sortlevel(1)['qux']
Out[799]:
L2
one -0.156479
two 0.819712
dtype: float64
```

On higher dimensional objects, you can sort any of the other axes by level if they have a MultiIndex:

```python
In [800]: df.T.sortlevel(1, axis=1)
Out[800]:
  zero   one   zero   one
     x     x     y     y
  0 -0.662357 -1.505397 -0.027793 0.307453
  1  2.725042  1.392009 -0.631023 -0.906534
```

The `MultiIndex` object has code to explicitly check the sort depth. Thus, if you try to index at a depth at which the index is not sorted, it will raise an exception. Here is a concrete example to illustrate this:

```python
In [801]: tuples = [('a', 'a'), ('a', 'b'), ('b', 'a'), ('b', 'b')]
In [802]: idx = MultiIndex.from_tuples(tuples)
In [803]: idx.lexsort_depth
Out[803]: 2
In [804]: reordered = idx[[1, 0, 3, 2]]
In [805]: reordered.lexsort_depth
Out[805]: 1
In [806]: s = Series(randn(4), index=reordered)
In [807]: s.ix['a':'a']
Out[807]:
  a   b
  -0.488326
  a  0.851918
dtype: float64
```

However:

```python
>>> s.ix[('a', 'b'):('b', 'a')]
Exception: MultiIndex lexsort depth 1, key was length 2
```
7.4.9 Swapping levels with swaplevel

The swaplevel function can switch the order of two levels:

```python
In [808]: df[:5]
Out[808]:
   0   1
one y  0.307453 -0.906534
     x -1.505397  1.392009
zero y -0.027793 -0.631023
     x -0.662357  2.725042
```

```python
In [809]: df[:5].swaplevel(0, 1, axis=0)
Out[809]:
   0   1
  y one  0.307453 -0.906534
     x one -1.505397  1.392009
  y zero -0.027793 -0.631023
     x zero -0.662357  2.725042
```

7.4.10 Reordering levels with reorder_levels

The reorder_levels function generalizes the swaplevel function, allowing you to permute the hierarchical index levels in one step:

```python
In [810]: df[:5].reorder_levels([1,0], axis=0)
Out[810]:
   0   1
  y one  0.307453 -0.906534
     x one -1.505397  1.392009
  y zero -0.027793 -0.631023
     x zero -0.662357  2.725042
```

7.4.11 Some gory internal details

Internally, the MultiIndex consists of a few things: the levels, the integer labels, and the level names:

```python
In [811]: index
Out[811]:
MultiIndex
[bar one, two, baz one, two, foo one, two, qux one, two]
```

```python
In [812]: index.levels
Out[812]: [Index([bar, baz, foo, qux], dtype=object), Index([one, two], dtype=object)]
```

```python
In [813]: index.labels
Out[813]: [array([0, 0, 1, 1, 2, 2, 3, 3]), array([0, 1, 0, 1, 0, 1, 0, 1])]
```

```python
In [814]: index.names
Out[814]: ['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.
7.5 Adding an index to an existing DataFrame

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.

7.5.1 Add an index using DataFrame columns

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, indexed DataFrame:

```
In [815]: data
Out[815]:
   a  b  c  d
0  bar one  z  1
1  bar two  y  2
2  foo one  x  3
3  foo two  w  4

In [816]: indexed1 = data.set_index('c')

In [817]: indexed1
Out[817]:
   a   b   d
    c
   z bar one  1
   y bar two  2
   x foo one  3
   w foo two  4

In [818]: indexed2 = data.set_index(['a', 'b'])

In [819]: indexed2
Out[819]:
   c   d
   c a b
   z bar one  z  1
   y bar two  y  2
   x foo one  x  3
   w foo two  w  4
```

The `append` keyword option allow you to keep the existing index and append the given columns to a MultiIndex:

```
In [820]: frame = data.set_index('c', drop=False)

In [821]: frame = frame.set_index(['a', 'b'], append=True)

In [822]: frame
Out[822]:
   c   d
   c a b
   z bar one  z  1
   y bar two  y  2
   x foo one  x  3
   w foo two  w  4
```

Other options in `set_index` allow you not drop the index columns or to add the index in-place (without creating a new object):
In [823]: data.set_index('c', drop=False)
Out[823]:
   a  b  c  d
  c  
z  bar  one  z  1
y  bar  two  y  2
x  foo  one  x  3
w  foo  two  w  4

In [824]: data.set_index(['a', 'b'], inplace=True)
Out[824]:
   c  d
  a  b
  bar  one  z  1
two  y  2
foo  one  x  3
two  w  4

In [825]: data
Out[825]:
   c  d
  a  b
  bar  one  z  1
two  y  2
foo  one  x  3
two  w  4

7.5.2 Remove / reset the index, reset_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 to set_index

In [826]: data
Out[826]:
   c  d
  a  b
  bar  one  z  1
two  y  2
foo  one  x  3
two  w  4

In [827]: data.reset_index()
Out[827]:
   a  b  c  d
  0  bar  one  z  1
  1  bar  two  y  2
  2  foo  one  x  3
  3  foo  two  w  4

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 [828]: frame
Out[828]:
   c  d
  c  a  b

7.5. Adding an index to an existing DataFrame
z bar one  z 1
y bar two  y 2
x foo one  x 3
w foo two  w 4

In [829]: frame.reset_index(level=1)
Out[829]:
   a  c  d
   c  b
z one  bar  z 1
y two  bar  y 2
x one  foo  x 3
w two  foo  w 4

reset_index takes an optional parameter drop which if true simply discards the index, instead of putting index values in the DataFrame’s columns.

Note: The reset_index method used to be called delevel which is now deprecated.

7.5.3 Adding an ad hoc index

If you create an index yourself, you can just assign it to the index field:

data.index = index

7.6 Indexing internal details

Note: The following is largely relevant for those actually working on the pandas codebase. And the source code is still the best place to look at the specifics of how things are implemented.

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
- **MultiIndex**: the standard hierarchical index object
- **date_range**: fixed frequency date range generated from a time rule or DateOffset. An ndarray of Python datetime objects

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
- **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
8.1 Statistical functions

8.1.1 Percent Change

Both Series and DataFrame has a method `pct_change` to compute the percent change over a given number of periods (using `fill_method` to fill NA/null values).

```python
In [209]: ser = Series(randn(8))

In [210]: ser.pct_change()
Out[210]:
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
```

```python
In [211]: df = DataFrame(randn(10, 4))

In [212]: df.pct_change(periods=3)
Out[212]:
       0         1          2         3
0   NaN       NaN       NaN       NaN
1   NaN       NaN       NaN       NaN
2  -0.218320 -1.054001  1.987147 -0.510183
3  -0.439121 -1.816454  0.649715 -4.822809
4  -0.127833 -3.042065 -5.866604 -1.776977
5  -2.596833 -1.959538 -2.111697 -3.798900
6  -0.117826 -2.169058  0.036094 -0.067696
7   2.492606 -1.357320 -1.205802 -1.558697
8  -1.012977  2.324558 -1.003744 -0.371806
9   2.492606 -1.357320 -1.205802 -1.558697
```

8.1.2 Covariance

The Series object has a method `cov` to compute covariance between series (excluding NA/null values).
In [213]: s1 = Series(randn(1000))
In [214]: s2 = Series(randn(1000))
In [215]: s1.cov(s2)
Out[215]: 0.00068010881743109321

Analogously, DataFrame has a method `cov` to compute pairwise covariances among the series in the DataFrame, also excluding NA/null values.

In [216]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [217]: frame.cov()
Out[217]:
         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 [218]: frame = DataFrame(randn(20, 3), columns=['a', 'b', 'c'])
In [219]: frame.ix[:5, 'a'] = np.nan
In [220]: frame.ix[5:10, 'b'] = np.nan
In [221]: frame.cov()
Out[221]:
         a        b        c
a  1.210090 -0.430629  0.018002
b -0.430629  1.240960  0.347188
c  0.018002  0.347188  1.301149

In [222]: frame.cov(min_periods=12)
Out[222]:
         a        b        c
a  1.210090   NaN     0.018002
b   NaN  1.240960  0.347188
c  0.018002  0.347188  1.301149

8.1.3 Correlation

Several methods for computing correlations are provided. Several kinds of correlation methods are provided:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pearson</code> (default)</td>
<td>Standard correlation coefficient</td>
</tr>
<tr>
<td><code>kendall</code></td>
<td>Kendall Tau correlation coefficient</td>
</tr>
<tr>
<td><code>spearman</code></td>
<td>Spearman rank correlation coefficient</td>
</tr>
</tbody>
</table>

All of these are currently computed using pairwise complete observations.

In [223]: frame = DataFrame(randn(1000, 5), columns=['a', 'b', 'c', 'd', 'e'])
In [224]: frame.ix[:,2:] = np.nan
# Series with Series

In [225]: frame['a'].corr(frame['b'])
Out[225]: 0.013479040400098763

In [226]: frame['a'].corr(frame['b'], method='spearman')
Out[226]: -0.0072898851595406388

# Pairwise correlation of DataFrame columns

In [227]: frame.corr()
Out[227]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.000000</td>
<td>0.013479</td>
<td>-0.049269</td>
<td>-0.042239</td>
<td>-0.028525</td>
</tr>
<tr>
<td>b</td>
<td>0.013479</td>
<td>1.000000</td>
<td>-0.020433</td>
<td>-0.011139</td>
<td>0.005654</td>
</tr>
<tr>
<td>c</td>
<td>-0.049269</td>
<td>-0.020433</td>
<td>1.000000</td>
<td>0.018587</td>
<td>-0.054269</td>
</tr>
<tr>
<td>d</td>
<td>-0.042239</td>
<td>-0.011139</td>
<td>0.018587</td>
<td>1.000000</td>
<td>-0.017060</td>
</tr>
<tr>
<td>e</td>
<td>-0.028525</td>
<td>0.005654</td>
<td>-0.054269</td>
<td>-0.017060</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

Note that non-numeric columns will be automatically excluded from the correlation calculation.

Like `cov`, `corr` also supports the optional `min_periods` keyword:

In [228]: frame = DataFrame(randn(20, 3), columns=['a', 'b', 'c'])

In [229]: frame.ix[:5, 'a'] = np.nan

In [230]: frame.ix[5:10, 'b'] = np.nan

In [231]: frame.corr()
Out[231]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.000000</td>
<td>-0.076520</td>
<td>0.160092</td>
</tr>
<tr>
<td>b</td>
<td>-0.076520</td>
<td>1.000000</td>
<td>0.135967</td>
</tr>
<tr>
<td>c</td>
<td>0.160092</td>
<td>0.135967</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

In [232]: frame.corr(min_periods=12)
Out[232]:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1.000000</td>
<td>NaN</td>
<td>0.160092</td>
</tr>
<tr>
<td>b</td>
<td>NaN</td>
<td>1.000000</td>
<td>0.135967</td>
</tr>
<tr>
<td>c</td>
<td>0.160092</td>
<td>0.135967</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

A related method `corrwith` is implemented on `DataFrame` to compute the correlation between like-labeled Series contained in different DataFrame objects.

In [233]: index = ['a', 'b', 'c', 'd', 'e']

In [234]: columns = ['one', 'two', 'three', 'four']

In [235]: df1 = DataFrame(randn(5, 4), index=index, columns=columns)

In [236]: df2 = DataFrame(randn(4, 4), index=index[:4], columns=columns)

In [237]: df1.corrwith(df2)
Out[237]:

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th>two</th>
<th>three</th>
<th>four</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
<td>-0.125501</td>
<td>-0.493244</td>
<td>0.344056</td>
<td>0.004183</td>
</tr>
<tr>
<td>two</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>three</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>four</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dtype: float64

8.1. Statistical functions 155
8.1.4 Data ranking

The `rank` method produces a data ranking with ties being assigned the mean of the ranks (by default) for the group:

```python
In [239]: s = Series(np.random.randn(5), index=list('abcde'))
In [240]: s['d'] = s['b']  # so there's a tie
In [241]: s.rank()
```

```
Out[241]:
a 5.0
b 2.5
c 1.0
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.

```python
In [242]: df = DataFrame(np.random.randn(10, 6))
In [244]: df.rank(1)
```

```
Out[244]:
     0    1     2    3     4     5
0 -0.904948 -1.63537 -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  0.463347
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 -1.056043  0.419477  NaN  1.375064
```

```python
In [245]: df['d'].rank()
```

```
Out[245]:
  0  1  2  3  4  5
0  1  2  3  4  5
1  2  3  1   5   3
2  3  4  2  2  1
3  4  5  2  4  5
4  5  6  3  5  3
5  5  6  3  5  3
6  6  7  4  5  3
7  7  8  5  7  5
8  8  9  6  8  6
9  9 10  7  9  7
```

Chapter 8. Computational tools
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

Note: These methods are significantly faster (around 10-20x) than `scipy.stats.rankdata`.

### 8.2 Moving (rolling) statistics / moments

For working with time series data, a number of functions are provided for computing common moving or rolling statistics. Among these are count, sum, mean, median, correlation, variance, covariance, standard deviation, skewness, and kurtosis. All of these methods are in the `pandas` namespace, but otherwise they can be found in `pandas.stats.moments`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rolling_count</code></td>
<td>Number of non-null observations</td>
</tr>
<tr>
<td><code>rolling_sum</code></td>
<td>Sum of values</td>
</tr>
<tr>
<td><code>rolling_mean</code></td>
<td>Mean of values</td>
</tr>
<tr>
<td><code>rolling_median</code></td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td><code>rolling_min</code></td>
<td>Minimum</td>
</tr>
<tr>
<td><code>rolling_max</code></td>
<td>Maximum</td>
</tr>
<tr>
<td><code>rolling_std</code></td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td><code>rolling_var</code></td>
<td>Unbiased variance</td>
</tr>
<tr>
<td><code>rolling_skew</code></td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td><code>rolling_kurt</code></td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td><code>rolling_quantile</code></td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td><code>rolling_apply</code></td>
<td>Generic apply</td>
</tr>
<tr>
<td><code>rolling_cov</code></td>
<td>Unbiased covariance (binary)</td>
</tr>
<tr>
<td><code>rolling_corr</code></td>
<td>Correlation (binary)</td>
</tr>
<tr>
<td><code>rolling_corr_pairwise</code></td>
<td>Pairwise correlation of DataFrame columns</td>
</tr>
<tr>
<td><code>rolling_window</code></td>
<td>Moving window function</td>
</tr>
</tbody>
</table>

Generally these methods all have the same interface. The binary operators (e.g. `rolling_corr`) take two Series or DataFrames. Otherwise, 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)
- `freq`: optionally specify a frequency string or `DateOffset` to pre-conform the data to. Note that prior to pandas v0.8.0, a keyword argument `time_rule` was used instead of `freq` that referred to the legacy time rule constants

These functions can be applied to ndarrays or Series objects:
In [246]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [247]: ts = ts.cumsum()

In [248]: ts.plot(style='k--')
Out[248]: <matplotlib.axes.AxesSubplot at 0x6691590>

In [249]: rolling_mean(ts, 60).plot(style='k')
Out[249]: <matplotlib.axes.AxesSubplot at 0x6691590>

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 [250]: df = DataFrame(randn(1000, 4), index=ts.index,
       ....:         columns=['A', 'B', 'C', 'D'])
       ....:

In [251]: df = df.cumsum()

In [252]: rolling_sum(df, 60).plot(subplots=True)
Out[252]:
array([[Axes(0.125, 0.747826; 0.775x0.152174),
        Axes(0.125, 0.565217; 0.775x0.152174),
        Axes(0.125, 0.382609; 0.775x0.152174), Axes(0.125, 0.2; 0.775x0.152174)], dtype=object)
The `rolling_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:

```python
In [253]: mad = lambda x: np.fabs(x - x.mean()).mean()
```

```python
In [254]: rolling_apply(ts, 60, mad).plot(style='k')
Out[254]: <matplotlib.axes.AxesSubplot at 0x6d4f750>
```

The `rolling_window` function performs a generic rolling window computation on the input data. The weights used in the window are specified by the `win_type` keyword. The list of recognized types are:

- boxcar
- triang
- blackman

8.2. Moving (rolling) statistics / moments
In [255]: ser = Series(randn(10), index=date_range('1/1/2000', periods=10))

In [256]: rolling_window(ser, 5, 'triang')
Out[256]:
2000-01-01      NaN
2000-01-02      NaN
2000-01-03      NaN
2000-01-04      NaN
2000-01-05  -0.622722
2000-01-06  -0.460623
2000-01-07  -0.229918
2000-01-08  -0.237308
2000-01-09  -0.335064
2000-01-10 -0.403449
Freq: D, dtype: float64

Note that the boxcar window is equivalent to rolling_mean:

In [257]: rolling_window(ser, 5, 'boxcar')
Out[257]:
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 [258]: rolling_mean(ser, 5)
Out[258]:
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
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 [259]: rolling_window(ser, 5, 'gaussian', std=0.1)
Out[259]:
2000-01-01 NaN
2000-01-02 NaN
2000-01-03 NaN
2000-01-04 NaN
2000-01-05 -0.261998
2000-01-06 -0.230600
2000-01-07  0.121276
2000-01-08 -0.136220
2000-01-09 -0.057945
2000-01-10 -0.199326
Freq: D, dtype: float64

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. This keyword is available in other rolling functions as well.

In [260]: rolling_window(ser, 5, 'boxcar')
Out[260]:
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 [261]: rolling_window(ser, 5, 'boxcar', center=True)
Out[261]:
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

In [262]: rolling_mean(ser, 5, center=True)
Out[262]:
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

8.2. Moving (rolling) statistics / moments
8.2.1 Binary rolling moments

rolling_cov and rolling_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: compute statistic for matching column names, returning a DataFrame

For example:

In [263]: df2 = df[:20]

In [264]: rolling_corr(df2, df2['B'], window=5)

Out[264]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.262853</td>
<td>1</td>
<td>0.334449</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.083745</td>
<td>1</td>
<td>-0.521587</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.292940</td>
<td>1</td>
<td>-0.658532</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.840416</td>
<td>1</td>
<td>0.796505</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>-0.135275</td>
<td>1</td>
<td>0.753895</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-0.346229</td>
<td>1</td>
<td>-0.682323</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>-0.365524</td>
<td>1</td>
<td>-0.775831</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.204761</td>
<td>1</td>
<td>-0.855874</td>
</tr>
<tr>
<td>2000-01-13</td>
<td>0.575218</td>
<td>1</td>
<td>-0.747531</td>
</tr>
<tr>
<td>2000-01-14</td>
<td>0.519499</td>
<td>1</td>
<td>-0.687277</td>
</tr>
<tr>
<td>2000-01-15</td>
<td>0.048982</td>
<td>1</td>
<td>0.167669</td>
</tr>
<tr>
<td>2000-01-16</td>
<td>0.217190</td>
<td>1</td>
<td>0.167564</td>
</tr>
<tr>
<td>2000-01-17</td>
<td>0.641180</td>
<td>1</td>
<td>-0.164780</td>
</tr>
<tr>
<td>2000-01-18</td>
<td>0.130422</td>
<td>1</td>
<td>0.322833</td>
</tr>
<tr>
<td>2000-01-19</td>
<td>0.317278</td>
<td>1</td>
<td>0.384528</td>
</tr>
<tr>
<td>2000-01-20</td>
<td>0.293598</td>
<td>1</td>
<td>0.159538</td>
</tr>
</tbody>
</table>

8.2.2 Computing rolling pairwise correlations

In financial data analysis and other fields it’s common to compute correlation matrices for a collection of time series. More difficult is to compute a moving-window correlation matrix. This can be done using the rolling_corr_pairwise function, which yields a Panel whose items are the dates in question:

In [265]: correls = rolling_corr_pairwise(df, 50)

In [266]: correls[df.index[-50]]

Out[266]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>-0.262853</td>
<td>1</td>
<td>0.334449</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.083745</td>
<td>1</td>
<td>-0.521587</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>-0.292940</td>
<td>1</td>
<td>-0.658532</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>0.840416</td>
<td>1</td>
<td>0.796505</td>
</tr>
<tr>
<td>2000-01-09</td>
<td>-0.135275</td>
<td>1</td>
<td>0.753895</td>
</tr>
<tr>
<td>2000-01-10</td>
<td>-0.346229</td>
<td>1</td>
<td>-0.682323</td>
</tr>
<tr>
<td>2000-01-11</td>
<td>-0.365524</td>
<td>1</td>
<td>-0.775831</td>
</tr>
<tr>
<td>2000-01-12</td>
<td>-0.204761</td>
<td>1</td>
<td>-0.855874</td>
</tr>
<tr>
<td>2000-01-13</td>
<td>0.575218</td>
<td>1</td>
<td>-0.747531</td>
</tr>
<tr>
<td>2000-01-14</td>
<td>0.519499</td>
<td>1</td>
<td>-0.687277</td>
</tr>
<tr>
<td>2000-01-15</td>
<td>0.048982</td>
<td>1</td>
<td>0.167669</td>
</tr>
<tr>
<td>2000-01-16</td>
<td>0.217190</td>
<td>1</td>
<td>0.167564</td>
</tr>
<tr>
<td>2000-01-17</td>
<td>0.641180</td>
<td>1</td>
<td>-0.164780</td>
</tr>
<tr>
<td>2000-01-18</td>
<td>0.130422</td>
<td>1</td>
<td>0.322833</td>
</tr>
<tr>
<td>2000-01-19</td>
<td>0.317278</td>
<td>1</td>
<td>0.384528</td>
</tr>
<tr>
<td>2000-01-20</td>
<td>0.293598</td>
<td>1</td>
<td>0.159538</td>
</tr>
</tbody>
</table>
8.3 Expanding window moment functions

A common alternative to rolling statistics is to use an expanding window, which yields the value of the statistic with all the data available up to that point in time. As these calculations are a special case of rolling statistics, they are implemented in pandas such that the following two calls are equivalent:

```python
In [268]: rolling_mean(df, window=len(df), min_periods=1)[:5]
Out[268]:
   A    B    C    D
2000-01-01 -1.388345 3.317290 0.344542 -0.036968
2000-01-02 -1.123132 3.622300 1.675867 0.595300
2000-01-03 -0.628502 3.626503 2.455240 1.060158
2000-01-04 -0.768740 3.888917 2.451354 1.281874
2000-01-05 -0.824034 4.108035 2.556112 1.140723
```

```python
In [269]: expanding_mean(df)[:5]
Out[269]:
   A    B    C    D
2000-01-01 -1.388345 3.317290 0.344542 -0.036968
2000-01-02 -1.123132 3.622300 1.675867 0.595300
2000-01-03 -0.628502 3.626503 2.455240 1.060158
2000-01-04 -0.768740 3.888917 2.451354 1.281874
2000-01-05 -0.824034 4.108035 2.556112 1.140723
```

8.3. Expanding window moment functions
Like the `rolling_` functions, the following methods are included in the `pandas` namespace or can be located in `pandas.stats.moments`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expanding_count</td>
<td>Number of non-null observations</td>
</tr>
<tr>
<td>expanding_sum</td>
<td>Sum of values</td>
</tr>
<tr>
<td>expanding_mean</td>
<td>Mean of values</td>
</tr>
<tr>
<td>expanding_median</td>
<td>Arithmetic median of values</td>
</tr>
<tr>
<td>expanding_min</td>
<td>Minimum</td>
</tr>
<tr>
<td>expanding_max</td>
<td>Maximum</td>
</tr>
<tr>
<td>expanding_std</td>
<td>Unbiased standard deviation</td>
</tr>
<tr>
<td>expanding_var</td>
<td>Unbiased variance</td>
</tr>
<tr>
<td>expanding_skew</td>
<td>Unbiased skewness (3rd moment)</td>
</tr>
<tr>
<td>expanding_kurt</td>
<td>Unbiased kurtosis (4th moment)</td>
</tr>
<tr>
<td>expanding_quantile</td>
<td>Sample quantile (value at %)</td>
</tr>
<tr>
<td>expanding_apply</td>
<td>Generic apply</td>
</tr>
<tr>
<td>expanding_cov</td>
<td>Unbiased covariance (binary)</td>
</tr>
<tr>
<td>expanding_corr</td>
<td>Correlation (binary)</td>
</tr>
<tr>
<td>expanding_corr_pairwise</td>
<td>Pairwise correlation of DataFrame columns</td>
</tr>
</tbody>
</table>

Aside from not having a `window` parameter, these functions have the same interfaces as their `rolling_` counterpart. 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.
- **freq**: optionally specify a *frequency string* or *DateOffset* to pre-conform the data to. Note that prior to pandas v0.8.0, a keyword argument `time_rule` was used instead of `freq` that referred to the legacy time rule constants.

**Note:** The output of the `rolling_` and `expanding_` functions do not return a NaN if there are at least `min_periods` non-null values in the current window. This differs from `cumsum`, `cumprod`, `cummax`, and `cummin`, which return NaN in the output wherever a NaN is encountered in the input.

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 `expanding_mean` output for the previous time series dataset:

```python
In [270]: ts.plot(style='k--')
Out[270]: <matplotlib.axes.AxesSubplot at 0x74323d0>

In [271]: expanding_mean(ts).plot(style='k')
Out[271]: <matplotlib.axes.AxesSubplot at 0x74323d0>
```
8.4 Exponentially weighted moment functions

A related set of functions are exponentially weighted versions of many of the above statistics. A number of EW (exponentially weighted) functions are provided using the blending method. For example, where \( y_t \) is the result and \( x_t \) the input, we compute an exponentially weighted moving average as

\[
y_t = \alpha y_{t-1} + (1 - \alpha) x_t
\]

One must have \( 0 < \alpha \leq 1 \), but rather than pass \( \alpha \) directly, it’s easier to think about either the span or center of mass (com) of an EW moment:

\[
\alpha = \begin{cases} \frac{2}{s+1}, & s = \text{span} \\ \frac{1}{c+1}, & c = \text{center of mass} \end{cases}
\]

You can pass one or the other to these functions but not both. Span corresponds to what is commonly called a “20-day EW moving average” for example. Center of mass has a more physical interpretation. For example, span = 20 corresponds to com = 9.5. Here is the list of functions available:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ewma</td>
<td>EW moving average</td>
</tr>
<tr>
<td>ewmvar</td>
<td>EW moving variance</td>
</tr>
<tr>
<td>ewmstd</td>
<td>EW moving standard deviation</td>
</tr>
<tr>
<td>ewmcorr</td>
<td>EW moving correlation</td>
</tr>
<tr>
<td>ewmcov</td>
<td>EW moving covariance</td>
</tr>
</tbody>
</table>

Here are an example for a univariate time series:

In [272]: plt.close('all')

In [273]: ts.plot(style='k--')
Out[273]: <matplotlib.axes.AxesSubplot at 0x77b2950>
In [274]: ewma(ts, span=20).plot(style='k')
Out[274]: <matplotlib.axes.AxesSubplot at 0x77b2950>

Note: The EW functions perform a standard adjustment to the initial observations whereby if there are fewer observations than called for in the span, those observations are reweighted accordingly.

8.5 Linear and panel regression

Note: We plan to move this functionality to statsmodels for the next release. Some of the result attributes may change names in order to foster naming consistency with the rest of statsmodels. We will provide every effort to provide compatibility with older versions of pandas, however.

We have implemented a very fast set of moving-window linear regression classes in pandas. Two different types of regressions are supported:

- Standard ordinary least squares (OLS) multiple regression
- Multiple regression (OLS-based) on panel data including with fixed-effects (also known as entity or individual effects) or time-effects.

Both kinds of linear models are accessed through the `ols` function in the pandas namespace. They all take the following arguments to specify either a static (full sample) or dynamic (moving window) regression:

- `window_type`: ‘full sample’ (default), ‘expanding’, or ‘rolling’
- `window`: size of the moving window in the `window_type='rolling'` case. If `window` is specified, `window_type` will be automatically set to ‘rolling’
- `min_periods`: minimum number of time periods to require to compute the regression coefficients
Generally speaking, the `ols` works by being given a \( y \) (response) object and an \( x \) (predictors) object. These can take many forms:

- \( y \): a Series, ndarray, or DataFrame (panel model)
- \( x \): Series, DataFrame, dict of Series, dict of DataFrame or Panel

Based on the types of \( y \) and \( x \), the model will be inferred to either a panel model or a regular linear model. If the \( y \) variable is a DataFrame, the result will be a panel model. In this case, the \( x \) variable must either be a Panel, or a dict of DataFrame (which will be coerced into a Panel).

### 8.5.1 Standard OLS regression

Let’s pull in some sample data:

```python
In [275]: from pandas.io.data import DataReader

In [276]: symbols = ['MSFT', 'GOOG', 'AAPL']

In [277]: data = dict((sym, DataReader(sym, "yahoo"))
    for sym in symbols)

In [278]: panel = Panel(data).swapaxes('items', 'minor')

In [279]: close_px = panel['Close']

# convert closing prices to returns
In [280]: rets = close_px / close_px.shift(1) - 1

In [281]: rets.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 830 entries, 2010-01-04 00:00:00 to 2013-04-22 00:00:00
Data columns:
AAPL 829 non-null values
GOOG 829 non-null values
MSFT 829 non-null values
dtypes: float64(3)

Let’s do a static regression of AAPL returns on GOOG returns:

```python
In [282]: model = ols(y=rets['AAPL'], x=rets.ix[:, ['GOOG']])

In [283]: model
```

```
Out[283]:
-------------------------Summary of Regression Analysis-------------------------
Formula: Y ~ <GOOG> + <intercept>
Number of Observations: 829
Number of Degrees of Freedom: 2
R-squared: 0.2372
Adj R-squared: 0.2363
Rmse: 0.0157
F-stat (1, 827): 257.2205, p-value: 0.0000
Degrees of Freedom: model 1, resid 827
-----------------------Summary of Estimated Coefficients------------------------
<table>
<thead>
<tr>
<th>Variable</th>
<th>Coef</th>
<th>Std Err</th>
<th>t-stat</th>
<th>p-value</th>
<th>CI 2.5%</th>
<th>CI 97.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOOG</td>
<td>0.5267</td>
<td>0.0328</td>
<td>16.04</td>
<td>0.0000</td>
<td>0.4623</td>
<td>0.5910</td>
</tr>
<tr>
<td>intercept</td>
<td>0.0007</td>
<td>0.0005</td>
<td>1.25</td>
<td>0.2101</td>
<td>-0.0004</td>
<td>0.0018</td>
</tr>
</tbody>
</table>
```
If we had passed a Series instead of a DataFrame with the single `GOOG` column, the model would have assigned the generic name `x` to the sole right-hand side variable.

We can do a moving window regression to see how the relationship changes over time:

```python
In [285]: model = ols(y=rets['AAPL'], x=rets.ix[:, ['GOOG']], window=250)

In [286]: model.beta['GOOG'].plot()
```

It looks like there are some outliers rolling in and out of the window in the above regression, influencing the results. We could perform a simple winsorization at the 3 STD level to trim the impact of outliers:

```python
In [287]: winz = rets.copy()

In [288]: std_1year = rolling_std(rets, 250, min_periods=20)

In [289]: cap_level = 3 * np.sign(winz) * std_1year

In [290]: winz[np.abs(winz) > 3 * std_1year] = cap_level
```
In [291]: winz_model = ols(y=winz[‘AAPL’], x=winz.ix[:, [‘GOOG’]],
.....: window=250)
.....:

In [292]: model.beta[‘GOOG’].plot(label="With outliers")
Out[292]: <matplotlib.axes.AxesSubplot at 0x8d91dd0>

In [293]: winz_model.beta[‘GOOG’].plot(label="Winsorized"); plt.legend(loc=’best’)
Out[293]: <matplotlib.legend.Legend at 0x8eb97d0>

So in this simple example we see the impact of winsorization is actually quite significant. Note the correlation after
winsorization remains high:

In [294]: winz.corrwith(rets)
Out[294]:
AAPL    0.988868
GOOG    0.973599
MSFT    0.998398
dtype: float64

Multiple regressions can be run by passing a DataFrame with multiple columns for the predictors $x$:

In [295]: ols(y=winz[‘AAPL’], x=winz.drop([‘AAPL’], axis=1))
Out[295]:
---Summary of Regression Analysis-------------------------
Formula: Y ~ <GOOG> + <MSFT> + <intercept>
Number of Observations: 829
Number of Degrees of Freedom: 3
R-squared: 0.3217
Adj R-squared: 0.3200
Rmse: 0.0140
F-stat (2, 826): 195.8485, p-value: 0.0000
Degrees of Freedom: model 2, resid 826

8.5. Linear and panel regression 169
### 8.5.2 Panel regression

We’ve implemented moving window panel regression on potentially unbalanced panel data (see this article if this means nothing to you). Suppose we wanted to model the relationship between the magnitude of the daily return and trading volume among a group of stocks, and we want to pool all the data together to run one big regression. This is actually quite easy:

```python
# make the units somewhat comparable
In [296]: volume = panel['Volume'] / 1e8

In [297]: model = ols(y=volume, x={'return' : np.abs(rets)})

In [298]: model
Out[298]:
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Coef</th>
<th>Std Err</th>
<th>t-stat</th>
<th>p-value</th>
<th>CI 2.5%</th>
<th>CI 97.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOOG</td>
<td>0.4636</td>
<td>0.0380</td>
<td>12.20</td>
<td>0.0000</td>
<td>0.3892</td>
<td>0.5381</td>
</tr>
<tr>
<td>MSFT</td>
<td>0.2956</td>
<td>0.0418</td>
<td>7.07</td>
<td>0.0000</td>
<td>0.2136</td>
<td>0.3777</td>
</tr>
<tr>
<td>intercept</td>
<td>0.0007</td>
<td>0.0005</td>
<td>1.38</td>
<td>0.1666</td>
<td>-0.0003</td>
<td>0.0016</td>
</tr>
</tbody>
</table>

---------------------------------End of Summary---------------------------------

In a panel model, we can insert dummy (0-1) variables for the “entities” involved (here, each of the stocks) to account the a entity-specific effect (intercept):

```python
In [299]: fe_model = ols(y=volume, x={'return' : np.abs(rets)},
                                 entity_effects=True)

```

```python
In [300]: fe_model
Out[300]:
```

<table>
<thead>
<tr>
<th>Variable</th>
<th>Coef</th>
<th>Std Err</th>
<th>t-stat</th>
<th>p-value</th>
<th>CI 2.5%</th>
<th>CI 97.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>3.4208</td>
<td>0.4670</td>
<td>7.32</td>
<td>0.0000</td>
<td>2.5055</td>
<td>4.3362</td>
</tr>
<tr>
<td>intercept</td>
<td>0.2227</td>
<td>0.0076</td>
<td>29.38</td>
<td>0.0000</td>
<td>0.2079</td>
<td>0.2376</td>
</tr>
</tbody>
</table>

---------------------------------End of Summary---------------------------------
Because we ran the regression with an intercept, one of the dummy variables must be dropped or the design matrix will not be full rank. If we do not use an intercept, all of the dummy variables will be included:

\textbf{In [301]:} fe_model = ols(y=volume, x={'return' : np.abs(rets)},
....: entity_effects=True, intercept=False)
....:

\textbf{In [302]:} fe_model
\textbf{Out[302]:}

\textbf{-------------------------Summary of Regression Analysis-------------------------}
Formula: Y ~ <return> + <FE_AAPL> + <FE_GOOG> + <FE_MSFT>
Number of Observations: 2487
Number of Degrees of Freedom: 4
R-squared: 0.7401
Adj R-squared: 0.7398
Rmse: 0.1368
F-stat (4, 2483): 2357.1701, p-value: 0.0000
Degrees of Freedom: model 3, resid 2483

\textbf{-----------------------Summary of Estimated Coefficients------------------------}

<table>
<thead>
<tr>
<th>Variable</th>
<th>Coef</th>
<th>Std Err</th>
<th>t-stat</th>
<th>p-value</th>
<th>CI 2.5%</th>
<th>CI 97.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>4.5616</td>
<td>0.2420</td>
<td>18.85</td>
<td>0.0000</td>
<td>4.0872</td>
<td>5.0360</td>
</tr>
<tr>
<td>FE_GOOG</td>
<td>-0.1540</td>
<td>0.0067</td>
<td>-22.87</td>
<td>0.0000</td>
<td>-0.1672</td>
<td>-0.1408</td>
</tr>
<tr>
<td>FE_MSFT</td>
<td>0.3873</td>
<td>0.0068</td>
<td>57.34</td>
<td>0.0000</td>
<td>0.3741</td>
<td>0.4006</td>
</tr>
<tr>
<td>intercept</td>
<td>0.1318</td>
<td>0.0057</td>
<td>23.04</td>
<td>0.0000</td>
<td>0.1206</td>
<td>0.1430</td>
</tr>
</tbody>
</table>

\textbf{-------------------End of Summary-------------------}

We can also include \textit{time effects}, which demeans the data cross-sectionally at each point in time (equivalent to including dummy variables for each date). More mathematical care must be taken to properly compute the standard errors in this case:

\textbf{In [303]:} te_model = ols(y=volume, x={'return' : np.abs(rets)},
....: time_effects=True, entity_effects=True)
....:

\textbf{In [304]:} te_model
\textbf{Out[304]:}

\textbf{-------------------------Summary of Regression Analysis-------------------------}
Formula: Y ~ <return> + <FE_GOOG> + <FE_MSFT>
Number of Observations: 2487
Number of Degrees of Freedom: 832
R-squared: 0.8159
Adj R-squared: 0.7235
Rmse: 0.1320
F-stat (3, 1655): 8.8284, p-value: 0.0000
Degrees of Freedom: model 831, resid 1655

\textbf{-----------------------Summary of Estimated Coefficients------------------------}

<table>
<thead>
<tr>
<th>Variable</th>
<th>Coef</th>
<th>Std Err</th>
<th>t-stat</th>
<th>p-value</th>
<th>CI 2.5%</th>
<th>CI 97.5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>3.7304</td>
<td>0.3422</td>
<td>10.90</td>
<td>0.0000</td>
<td>3.0597</td>
<td>4.4011</td>
</tr>
<tr>
<td>FE_GOOG</td>
<td>-0.1556</td>
<td>0.0065</td>
<td>-23.89</td>
<td>0.0000</td>
<td>-0.1684</td>
<td>-0.1428</td>
</tr>
</tbody>
</table>

\textbf{-------------------End of Summary-------------------}
Here the intercept (the mean term) is dropped by default because it will be 0 according to the model assumptions, having subtracted off the group means.

### 8.5.3 Result fields and tests

We’ll leave it to the user to explore the docstrings and source, especially as we’ll be moving this code into statsmodels in the near future.
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.

### 9.1 Missing data basics

#### 9.1.1 When / why does data become missing?

Some might quibble over our usage of *missing*. By “missing” we simply mean null 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 [1159]: df = DataFrame(randn(5, 3), index=['a', 'c', 'e', 'f', 'h'], 
    columns=['one', 'two', 'three'])
......: 
    columns=['one', 'two', 'three'])
......:

In [1160]: df['four'] = 'bar'
In [1161]: df['five'] = df['one'] > 0

In [1162]: df
Out[1162]:
   one  two  three  four  five
a  0.059117  1.138469 -2.400634  bar  True
b -0.280853  0.025653 -1.386071  bar  False
c  0.863937  0.252462  1.500571  bar  True
d  1.053202 -2.338595  -0.374279  bar  True
e  2.359958 -1.157886 -0.551865  bar  False

In [1163]: df2 = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

In [1164]: df2
Out[1164]:
   one  two  three  four  five
a  0.059117  1.138469 -2.400634  bar  True
b -0.280853  0.025653 -1.386071  bar  False
c  0.863937  0.252462  1.500571  bar  True
d  1.053202 -2.338595  -0.374279  bar  True
e  2.359958 -1.157886 -0.551865  bar  False
```
9.1.2 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 “null”.

Until recently, for legacy reasons inf and -inf were also considered to be “null” in computations. This is no longer the case by default; use the mode.use_inf_as_null option to recover it. To make detecting missing values easier (and across different array dtypes), pandas provides the isnull() and notnull() functions, which are also methods on Series objects:

```
In [1165]: df2['one']
Out[1165]:
a    0.059117
b   NaN
   ...  
h   -2.359958
Name: one, dtype: float64

In [1166]: isnull(df2['one'])
Out[1166]:
a   False
b  True
   ...  
h  True
Name: one, dtype: bool

In [1167]: df2['four'].notnull()
Out[1167]:
a   True
b  False
c   True
   ...  
h   True
dtype: bool
```
Summary: NaN and None (in object arrays) are considered missing by the isnull and notnull functions. inf and –inf are no longer considered missing by default.

9.2 Calculations with missing data

Missing values propagate naturally through arithmetic operations between pandas objects.

In [1168]: a
Out[1168]:
   one   two
  a 0.059117  1.138469
  b 0.059117  1.138469
  c -0.280853  0.025653
  d -0.280853  0.025653
  e  0.863937  0.252462

In [1169]: b
Out[1169]:
   one   two   three
  a  0.059117  1.138469 -2.400634
  b  NaN        NaN        NaN
  c -0.280853  0.025653 -1.386071
  d  NaN        NaN        NaN
  e  0.863937  0.252462  1.500571

In [1170]: a + b
Out[1170]:
   one  three   two
  a 0.118234  NaN  2.276938
  b  NaN      NaN  NaN
  c -0.561707  NaN -1.386071
  d  NaN      NaN  NaN
  e  1.727874  NaN  0.504923

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 NA
• Methods like cumsum and cumprod ignore NA values, but preserve them in the resulting arrays

In [1171]: df
Out[1171]:
   one   two   three
  a 0.059117  1.138469 -2.400634
  b  NaN      NaN        NaN
  c -0.280853  0.025653 -1.386071
  d  NaN      NaN        NaN
  e  0.863937  0.252462  1.500571
  f  1.053202 -2.338595 -0.374279
  g  NaN      NaN        NaN
  h -2.359958 -1.157886 -0.551865

In [1172]: df['one'].sum()
Out[1172]: -0.66455558290247652
In [1173]: df.mean(1)
Out[1173]:
a  -0.401016
b  NaN
c  -0.547090
d  NaN
e  0.872323
f  -0.553224
g  NaN
h  -1.356570
dtype: float64

In [1174]: df.cumsum()
Out[1174]:
     one   two   three
a  0.059117 1.138469 -2.400634
b   NaN   NaN   NaN
c -0.221736 1.164122 -3.786705
d   NaN   NaN   NaN
e  0.642200 1.416584 -2.286134
f  1.695403 -0.922011 -2.660413
g   NaN   NaN   NaN
h -0.664556 -2.079897 -3.212278

9.2.1 NA values in GroupBy

NA groups in GroupBy are automatically excluded. This behavior is consistent with R, for example.

9.3 Cleaning / filling missing data

pandas objects are equipped with various data manipulation methods for dealing with missing data.

9.3.1 Filling missing values: fillna

The fillna function can “fill in” NA values with non-null data in a couple of ways, which we illustrate:

Replace NA with a scalar value

In [1175]: df2
Out[1175]:
    one   two   three   four   five
a  0.059117 1.138469 -2.400634  bar  True
b   NaN   NaN   NaN   NaN   NaN
c -0.280853 0.025653 -1.386071  bar False
d   NaN   NaN   NaN   NaN   NaN
e  0.863937 0.252462  1.500571  bar  True
f  1.053202 -2.338595 -0.374279  bar  True
g   NaN   NaN   NaN   NaN   NaN
h -2.359958 -1.157886 -0.551865  bar False

In [1176]: df2.fillna(0)
Out[1176]:
    one   two   three   four   five
a  0.059117 1.138469 -2.400634  bar  True
In [1177]: df2[‘four’].fillna(‘missing’)
Out[1177]:
   a   bar
  b missing
  c   bar
  d missing
  e   bar
  f   bar
  g missing
  h   bar
Name: four, dtype: object

Fill gaps forward or backward

Using the same filling arguments as reindexing, we can propagate non-null values forward or backward:

In [1178]: df
Out[1178]:
    one    two    three
   a 0.059117  1.138469 -2.400634
   b     NaN     NaN     NaN
   c -0.280853  0.025653 -1.386071
   d     NaN     NaN     NaN
   e  0.863937  0.252462  1.500571
   f  1.053202 -2.338595 -0.374279
   g     NaN     NaN     NaN
   h -2.359958 -1.157886 -0.551865

In [1179]: df.fillna(method=’pad’)
Out[1179]:
    one    two    three
   a 0.059117  1.138469 -2.400634
   b 0.059117  1.138469 -2.400634
   c -0.280853  0.025653 -1.386071
   d -0.280853  0.025653 -1.386071
   e  0.863937  0.252462  1.500571
   f  1.053202 -2.338595 -0.374279
   g  1.053202 -2.338595 -0.374279
   h -2.359958 -1.157886 -0.551865

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 [1180]: df
Out[1180]:
   one    two    three
   a 0.059117  1.138469 -2.400634
   b     NaN     NaN     NaN
   c     NaN     NaN     NaN
   d     NaN     NaN     NaN
   e  0.863937  0.252462  1.500571
In [1181]: df.fillna(method='pad', limit=1)
Out[1181]:
    one   two   three
a  0.059117  1.138469 -2.400634
b  0.059117  1.138469 -2.400634
c         NaN       NaN       NaN
d         NaN       NaN       NaN
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595  -0.374279
g  1.053202 -2.338595  -0.374279
h -2.359958 -1.157886  -0.551865

To remind you, these are the available filling methods:

<table>
<thead>
<tr>
<th>Method</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>pad / ffill</td>
<td>Fill values forward</td>
</tr>
<tr>
<td>bfill / backfill</td>
<td>Fill values backward</td>
</tr>
</tbody>
</table>

With time series data, using pad/ffill is extremely common so that the “last known value” is available at every time point.

### 9.3.2 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 the `dropna` method:

In [1182]: df
Out[1182]:
    one   two   three
a  0.059117  1.138469 -2.400634
b         NaN       NaN       NaN
c         NaN       NaN       NaN
d         NaN       NaN       NaN
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595  -0.374279
g         NaN       NaN       NaN
h -2.359958 -1.157886  -0.551865

In [1183]: df.dropna(axis=0)
Out[1183]:
    one   two   three
a  0.059117  1.138469 -2.400634
e  0.863937  0.252462  1.500571
f  1.053202 -2.338595  -0.374279
h -2.359958 -1.157886  -0.551865

In [1184]: df.dropna(axis=1)
Out[1184]:
    two   three
a  1.138469  -2.400634
b  0.000000   0.000000
c  0.000000   0.000000
d  0.000000   0.000000
e  0.252462  1.500571
f -2.338595  -0.374279
In [1185]: df[‘one’].dropna()
Out[1185]:
a  0.059117
e  0.863937
f  1.053202
h -2.359958
Name: one, dtype: float64

dropna is presently only implemented for Series and DataFrame, but will be eventually added to Panel. Series.dropna is a simpler method as it only has one axis to consider. DataFrame.dropna has considerably more options, which can be examined in the API.

9.3.3 Interpolation

A linear interpolate method has been implemented on Series. The default interpolation assumes equally spaced points.

In [1186]: ts.count()
Out[1186]: 61

In [1187]: ts.head()
Out[1187]:
2000-01-31  0.469112
2000-02-29   NaN
2000-03-31   NaN
2000-04-28   NaN
2000-05-31   NaN
Freq: BM, dtype: float64

In [1188]: ts.interpolate().count()
Out[1188]: 100

In [1189]: ts.interpolate().head()
Out[1189]:
2000-01-31  0.469112
2000-02-29  0.435428
2000-03-31  0.401743
2000-04-28  0.368059
2000-05-31  0.334374
Freq: BM, dtype: float64

In [1190]: ts.interpolate().plot()
Out[1190]: <matplotlib.axes.AxesSubplot at 0xeece490>
Index aware interpolation is available via the `method` keyword:

```
In [1191]: ts
Out[1191]:
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 [1192]: ts.interpolate()
Out[1192]:
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 [1193]: ts.interpolate(method='time')
Out[1193]:
2000-01-31    0.469112
2000-02-29    0.273272
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 [1194]: ser
Out[1194]:
0   0
1  NaN
10  10
dtype: float64
```

```
In [1195]: ser.interpolate()
Out[1195]:
0   0
1   5
10  10
dtype: float64
```

```
In [1196]: ser.interpolate(method='values')
Out[1196]:
0   0
1   1
10  10
dtype: float64
```

### 9.3.4 Replacing Generic Values

Often times we want to replace arbitrary values with other values. New in v0.8 is the `replace` method in `Series/DataFrame` that 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 [1197]: ser = Series([0., 1., 2., 3., 4.])
```

```
In [1198]: ser.replace(0, 5)
Out[1198]:
0   5
1   1
2   2
3   3
4   4
dtype: float64
```

You can replace a list of values by a list of other values:

```
In [1199]: ser.replace([0, 1, 2, 3, 4], [4, 3, 2, 1, 0])
Out[1199]:
0   4
1   3
2   2
3   1
4   0
dtype: float64
```

You can also specify a mapping dict:

```
In [1200]: ser.replace({0: 10, 1: 100})
Out[1200]:
0   10
1  100
```

### 9.3. Cleaning / filling missing data
For a DataFrame, you can specify individual values by column:

```
In [1201]: df = DataFrame({'a': [0, 1, 2, 3, 4], 'b': [5, 6, 7, 8, 9]})
```

```
In [1202]: df.replace({'a': 0, 'b': 5}, 100)
Out[1202]:
   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 [1203]: ser.replace([1, 2, 3], method='pad')
```

```
0
1
2
3
4
 dtype: float64
```

### 9.4 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 reindexing will cause missing data to be introduced into, say, a Series or DataFrame. Here they are:

<table>
<thead>
<tr>
<th>data type</th>
<th>Cast to</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>float</td>
</tr>
<tr>
<td>boolean</td>
<td>object</td>
</tr>
<tr>
<td>float</td>
<td>no cast</td>
</tr>
<tr>
<td>object</td>
<td>no cast</td>
</tr>
</tbody>
</table>

For example:

```
In [1204]: s = Series(randn(5), index=[0, 2, 4, 6, 7])
```

```
In [1205]: s > 0
Out[1205]:
0  False
2   True
4   True
6   True
7   True
dtype: bool
```

```
In [1206]: (s > 0).dtype
Out[1206]: dtype('bool')
```

```
```
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 [1210]: reindexed = s.reindex(range(8)).fillna(0)

In [1211]: reindexed[crit]
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-1211-2da204ed1ac7> in <module>()
----> 1 reindexed[crit]
/home/wesm/code/pandasplus/pandas/core/series.pyc in __getitem__(self, key)
       622 # special handling of boolean data with NAs stored in object
       623 # arrays. Since we can’t represent NA with dtype=bool
--> 624 if _is_bool_indexer(key):
       625     key = _check_bool_indexer(self.index, key)
       626
/home/wesm/code/pandasplus/pandas/core/common.pyc in _is_bool_indexer(key)
       1070 if not lib.is_bool_array(key):
       1071     if isnull(key).any():
--> 1072         raise ValueError(’cannot index with vector containing ’
       1073             ’NA / NaN values’)
       1074     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 [1212]: reindexed[crit.fillna(False)]
Out[1212]:
2  1.314232
4  0.690579
6  0.995761
7  2.396780
dtype: float64
```

```
In [1213]: reindexed[crit.fillna(True)]
Out[1213]:
1  0.000000
2  1.314232
3  0.000000
4  0.690579
5  0.000000
```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0.995761</td>
</tr>
<tr>
<td>7</td>
<td>2.396780</td>
</tr>
<tr>
<td></td>
<td><strong>dtype: float64</strong></td>
</tr>
</tbody>
</table>
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

Of these, the split step is the most straightforward. In fact, in many situations you may wish to split the data set into groups and do something with those groups yourself. In the apply step, we might wish to one of the following:

- **Aggregation**: computing a summary statistic (or statistics) about each group. Some examples:
  - Compute group sums or means
  - Compute group sizes / counts
- **Transformation**: perform some group-specific computations and return a like-indexed. Some examples:
  - Standardizing data (zscore) within group
  - Filling NAs within groups with a value derived from each group
- **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 method 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:

```sql
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.

### 10.1 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 do the following:

```python
>>> 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
- A list of any of the above things

Collectively we refer to the grouping objects as the keys. For example, consider the following DataFrame:

```
In [518]: df = DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
....:'foo', 'bar', 'foo', 'foo'],
....:'B' : ['one', 'one', 'two', 'three',
....:'two', 'two', 'one', 'three'],
....:'C' : randn(8), 'D' : randn(8)})
```

```
In [519]: df
Out[519]:
   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   three -0.173215  -0.706771
6  foo    one  0.119209  -1.039575
7  foo   three -1.044236   0.271860
```

We could naturally group by either the A or B columns or both:

```
In [520]: grouped = df.groupby(‘A’)
In [521]: grouped = df.groupby([‘A’, ‘B’])
```

These will split the DataFrame on its index (rows). We could also split by the columns:

```
In [522]: def get_letter_type(letter):
    if letter.lower() in ‘aeiou’:
        return ‘vowel’
    else:
        return ‘consonant’
In [523]: grouped = df.groupby(get_letter_type, axis=1)
```

Starting with 0.8, pandas Index objects now supports 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 [524]: lst = [1, 2, 3, 1, 2, 3]
In [525]: s = Series({1, 2, 3, 10, 20, 30}, lst)
In [526]: grouped = s.groupby(level=0)
In [527]: grouped.first()
```
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.

### 10.1.1 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:

```python
In [530]: df.groupby('A').groups
Out[530]: {'bar': [1, 3, 5], 'foo': [0, 2, 4, 6, 7]}
```

```python
In [531]: df.groupby(get_letter_type, axis=1).groups
Out[531]: {'consonant': ['B', 'C', 'D'], 'vowel': ['A']}
```

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:

```python
In [532]: grouped = df.groupby(['A', 'B'])
In [533]: grouped.groups
Out[533]:
{('bar', 'one'): [1],
 ('bar', 'three'): [3],
 ('bar', 'two'): [5],
 ('foo', 'one'): [0, 6],
 ('foo', 'three'): [7],
 ('foo', 'two'): [2, 4]}

In [534]: len(grouped)
Out[534]: 6
```

By default the group keys are sorted during the groupby operation. You may however pass `sort`````=``````False for potential speedups:

### 10.1. Splitting an object into groups
In [535]: df2 = DataFrame({'X': ['B', 'B', 'A', 'A'], 'Y': [1, 2, 3, 4]})

In [536]: df2.groupby(['X'], sort=True).sum()
Out[536]:
   Y
X  
A  7
B  3

In [537]: df2.groupby(['X'], sort=False).sum()
Out[537]:
   Y
X  
B  3
A  7

10.1.2 GroupBy with MultiIndex

With hierarchically-indexed data, it's quite natural to group by one of the levels of the hierarchy.

In [538]: s
Out[538]:
   first   second
bar    one   -0.424972
        two    0.567020
baz    one    0.276232
        two   -1.087401
foo    one   -0.673690
        two    0.113648
qux    one   -1.478427
        two    0.524988
dtype: float64

In [539]: grouped = s.groupby(level=0)

In [540]: grouped.sum()
Out[540]:
   first
bar   0.142048
baz  -0.811169
foo  -0.560041
qux  -0.953439
dtype: float64

If the MultiIndex has names specified, these can be passed instead of the level number:

In [541]: s.groupby(level='second').sum()
Out[541]:
   second
one   -2.300857
two    0.118256
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 [542]: s.sum(level='second')
Out[542]:
second
one   -2.300857
two    0.118256
dtype: float64

Also as of v0.6, grouping with multiple levels is supported.

In [543]: s
Out[543]:
   first  second  third
bar    doo   one    0.404705
       two    0.577046
baz    bee   one   -1.715002
       two   -1.039268
foo    bop   one   -0.370647
       two   -1.157892
qux    bop   one  -1.344312
       two    0.844885
dtype: float64

In [544]: s.groupby(level=['first','second']).sum()
Out[544]:
   first  second
bar doo  0.981751
baz bee -2.754270
foo bop -1.528539
qux bop -0.499427
dtype: float64

More on the `sum` function and aggregation later.

**10.1.3 DataFrame column selection in GroupBy**

Once you have created the `GroupBy` object from a DataFrame, for example, 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 [545]: grouped = df.groupby(["A"])

In [546]: grouped_C = grouped["C"]

In [547]: grouped_D = grouped["D"]

This is mainly syntactic sugar for the alternative and much more verbose:

In [548]: df["C"].groupby(df["A"])
Out[548]: <pandas.core.groupby.SeriesGroupBy at 0xb42d610>

Additionally this method avoids recomputing the internal grouping information derived from the passed key.

**10.2 Iterating through groups**

With the `GroupBy` object in hand, iterating through the grouped data is very natural and functions similarly to `itertools.groupby`:
In [549]: grouped = df.groupby('A')

In [550]: for name, group in grouped:
       ......: print name
       ......: print group
       ......:
bar
       A    B    C    D
1 bar one -0.282863 -2.104569
3 bar three -1.135632 1.071804
5 bar two -0.173215 -0.706771
foo
       A    B    C    D
0 foo one 0.469112 -0.861849
2 foo two -1.509059 -0.494929
4 foo two 1.212112 0.721555
6 foo one 0.119209 -1.039575
7 foo three -1.044236 0.271860

In the case of grouping by multiple keys, the group name will be a tuple:

In [551]: for name, group in df.groupby(['A', 'B']):
       ......: print name
       ......: print group
       ......:
('bar', 'one')
       A    B    C    D
1 bar one -0.282863 -2.104569
('bar', 'three')
       A    B    C    D
3 bar three -1.135632 1.071804
('bar', 'two')
       A    B    C    D
5 bar two -0.173215 -0.706771
('foo', 'one')
       A    B    C    D
0 foo one 0.469112 -0.861849
6 foo one 0.119209 -1.039575
('foo', 'three')
       A    B    C    D
7 foo three -1.044236 0.271860
('foo', 'two')
       A    B    C    D
2 foo two -1.509059 -0.494929
4 foo two 1.212112 0.721555

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:

10.3 Aggregation

Once the GroupBy object has been created, several methods are available to perform a computation on the grouped data. An obvious one is aggregation via the aggregate or equivalently agg method:

In [552]: grouped = df.groupby('A')

In [553]: grouped.aggregate(np.sum)
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 [556]: grouped = df.groupby(['A', 'B'], as_index=False)

In [557]: grouped.aggregate(np.sum)

Out[557]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>one</td>
<td>-0.282863</td>
<td>-2.104569</td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>three</td>
<td>-1.135632</td>
<td>1.071804</td>
</tr>
<tr>
<td>2</td>
<td>bar</td>
<td>two</td>
<td>-0.173215</td>
<td>-0.706771</td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>one</td>
<td>0.588321</td>
<td>-1.901424</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>three</td>
<td>-1.044236</td>
<td>0.271860</td>
</tr>
<tr>
<td>5</td>
<td>foo</td>
<td>two</td>
<td>-0.296946</td>
<td>0.226626</td>
</tr>
</tbody>
</table>

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 [558]: df.groupby('A', as_index=False).sum().reset_index()

Out[558]:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>-1.591710</td>
<td>-1.739537</td>
</tr>
<tr>
<td>1</td>
<td>foo</td>
<td>-0.752861</td>
<td>-1.402938</td>
</tr>
</tbody>
</table>

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 [560]: grouped.size()

Out[560]:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>191</td>
</tr>
</tbody>
</table>
10.3.1 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:

```python
In [561]: grouped = df.groupby('A')
In [562]: grouped['C'].agg([np.sum, np.mean, np.std])
Out[562]:
       sum   mean     std
A  
bar -1.591710 -0.530570  0.526860
foo -0.752861 -0.150572  1.113308
```

If a dict is passed, the keys will be used to name the columns. Otherwise the function’s name (stored in the function object) will be used.

```python
In [563]: grouped['D'].agg({'result1' : np.sum, 'result2' : np.mean})
```

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:

```python
In [564]: grouped.agg([np.sum, np.mean, np.std])
Out[564]:
       C          D
       sum   mean     std   sum   mean     std
A  
bar -1.591710 -0.530570  0.526860 -1.739537 -0.579846  1.591986
foo -0.752861 -0.150572  1.113308 -1.402938 -0.280588  0.753219
```

Passing a dict of functions has different behavior by default, see the next section.

10.3.2 Applying different functions to DataFrame columns

By passing a dict to `aggregate` you can apply a different aggregation to the columns of a DataFrame:

```python
In [565]: grouped.agg({'C' : np.sum, 'D' : lambda x: np.std(x, ddof=1)})
```

Passing a dict of functions has different behavior by default, see the next section.
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 [566]: grouped.agg({'C' : 'sum', 'D' : 'std'})
Out[566]:
     C     D
A bar -1.591710 1.591986
     foo -0.752861 0.753219
```

### 10.3.3 Cython-optimized aggregation functions

Some common aggregations, currently only `sum`, `mean`, and `std`, have optimized Cython implementations:

```
In [567]: df.groupby('A').sum()
Out[567]:
     C    D
A bar -1.591710 -1.739537
     foo -0.752861 -1.402938
In [568]: df.groupby(['A', 'B']).mean()
Out[568]:
     C    D
A B one -0.282863 -2.104569
     three -1.135632 1.071804
     two -0.173215 -0.706771
     foo one 0.294161 -0.950712
     three -1.044236 0.271860
     two -0.148473 0.113313
```

Of course `sum` and `mean` are implemented on pandas objects, so the above code would work even without the special versions via dispatching (see below).

### 10.4 Transformation

The `transform` method returns an object that is indexed the same (same size) as the one being grouped. Thus, the passed transform function should return a result that is the same size as the group chunk. For example, suppose we wished to standardize the data within each group:

```
In [569]: index = date_range('10/1/1999', periods=1100)
In [570]: ts = Series(np.random.normal(0.5, 2, 1100), index)
In [571]: ts = rolling_mean(ts, 100, 100).dropna()  
In [572]: ts.head()
Out[572]:
2000-01-08 0.536925
2000-01-09 0.494448
2000-01-10 0.496114
```

10.4. Transformation
In [574]: key = lambda x: x.year

In [575]: zscore = lambda x: (x - x.mean()) / x.std()

In [576]: transformed = ts.groupby(key).transform(zscore)

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 [577]: grouped = ts.groupby(key)

In [578]: grouped.mean()
Out[578]:
2000 0.416344
2001 0.416987
2002 0.599380
dtype: float64

In [579]: grouped.std()
Out[579]:
2000 0.174755
2001 0.309640
2002 0.266172
dtype: float64

# Transformed Data
In [580]: grouped_trans = transformed.groupby(key)

In [581]: grouped_trans.mean()
Out[581]:
2000 -3.122696e-16
2001 -2.688869e-16
2002 -1.499001e-16
dtype: float64

In [582]: grouped_trans.std()
Out[582]:
2000 1
2001 1
2002 1
dtype: float64

We can also visually compare the original and transformed data sets.
Another common data transform is to replace missing data with the group mean.

```python
In [585]: data_df
Out[585]:
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 0 to 999
Data columns:
A 908 non-null values
B 953 non-null values
C 820 non-null values
dtypes: float64(3)
```

```python
In [586]: countries = np.array(['US', 'UK', 'GR', 'JP'])
In [587]: key = countries[np.random.randint(0, 4, 1000)]
In [588]: grouped = data_df.groupby(key)
```

```python
# Non-NA count in each group
In [589]: grouped.count()
Out[589]:
      A  B  C
GR  219 223 194
JP  238 250 211
UK  228 239 213
US  223 241 202
```

```python
In [590]: f = lambda x: x.fillna(x.mean())
In [591]: transformed = grouped.transform(f)
```

We can verify that the group means have not changed in the transformed data and that the transformed data contains...
no NAs.

In [592]: grouped_trans = transformed.groupby(key)

In [593]: grouped.mean() # original group means
Out[593]:
          A     B     C
GR  0.093655 -0.004978 -0.049883
JP -0.067605  0.025828  0.006752
UK -0.054246  0.031742  0.068974
US  0.084334 -0.013433  0.056589

In [594]: grouped_trans.mean() # transformation did not change group means
Out[594]:
          A     B     C
GR  0.093655 -0.004978 -0.049883
JP -0.067605  0.025828  0.006752
UK -0.054246  0.031742  0.068974
US  0.084334 -0.013433  0.056589

In [595]: grouped.count() # original has some missing data points
Out[595]:
          A     B     C
GR   219  223  194
JP   238  250  211
UK   228  239  213
US   223  241  202

In [596]: grouped_trans.count() # counts after transformation
Out[596]:
          A     B     C
GR   234  234  234
JP   264  264  264
UK   251  251  251
US   251  251  251

In [597]: grouped_trans.size() # Verify non-NA count equals group size
Out[597]:
    GR    JP    UK    US
GR  234        264  251  251
dtype: int64

10.5 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 [598]: grouped = df.groupby('A')

In [599]: grouped.agg(lambda x: x.std())
Out[599]:
    B     C     D
A bar NaN   0.526860  1.591986
foo NaN  1.113308  0.753219
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 [600]: grouped.std()
Out[600]:
     C     D
A  0.52686  1.59198
bar foo  1.11331  0.75322
```

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 [601]: tsdf = DataFrame(randn(1000, 3),
                         index=date_range('1/1/2000', periods=1000),
                         columns=['A', 'B', 'C'])

In [602]: tsdf.ix[::2] = np.nan

In [603]: grouped = tsdf.groupby(lambda x: x.year)

In [604]: grouped.fillna(method='pad')
```

In this example, we chopped the collection of time series into yearly chunks then independently called fillna on the
groups.

### 10.6 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 [605]: df
Out[605]:
    A    B    C    D
 0 foo   one 0.469112 -0.861849
 1 bar   one -0.282863 -2.104569
 2 foo   two 1.409059 -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
```

10.6. Flexible apply
In [606]: grouped = df.groupby('A')

# could also just call .describe()
In [607]: grouped['C'].apply(lambda x: x.describe())
Out[607]:
    A
   bar   count  3.000000
      mean  -0.530570
       std   0.526860
      min  -1.135632
     25%  -0.709248
    50%  -0.282863
    75%  -0.228039
      max  -0.173215
   foo   count  5.000000
      mean  -0.150572
       std   1.113308
      min  -1.509059
     25%  -1.044236
    50%   0.119209
    75%   0.469112
      max   1.212112
dtype: float64

The dimension of the returned result can also change:
In [608]: grouped = df.groupby('A')['C']
In [609]: def f(group):
       ....:     return DataFrame({'original': group,
       ....:                           'demeaned': group - group.mean()})
       ....:

In [610]: grouped.apply(f)
Out[610]:
            demeaned       original
    0  0.619685          0.469112
    1  0.247707         -0.282863
    2 -1.358486         -1.509059
    3 -0.605062         -1.135632
    4  1.362684           1.212112
    5  0.357355          0.469112
    6  0.269781          0.119209
    7 -0.893664         -1.044236

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 [611]: def f(x):
       ....:     return Series([x, x**2], index=['x', 'x^2'])
       ....:

In [612]: s = Series(np.random.rand(5))
In [613]: s
Out[613]:
0  0.785887
1  0.498525
2  0.933703
In [614]: s.apply(f)
Out[614]:
          x   x^s
0  0.785887  0.617619
1  0.498525  0.248528
2  0.933703  0.871801
3  0.154106  0.023749
4  0.271779  0.073864

10.7 Other useful features

10.7.1 Automatic exclusion of “nuisance” columns

Again consider the example DataFrame we’ve been looking at:

In [615]: df
Out[615]:
   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  0.271860
7  foo three  1.044236  0.753219

Supposed we wished 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 [616]: df.groupby('A').std()
Out[616]:
   C     D
A
  bar  0.526860  1.591986
  foo  1.113308  0.753219

10.7.2 NA group handling

If there are any NaN values in the grouping key, these will be automatically excluded. So there will never be an “NA 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).

10.7.3 Grouping with ordered factors

Categorical variables represented as instance of pandas’s Factor class can be used as group keys. If so, the order of the levels will be preserved:
In [617]: data = Series(np.random.randn(100))

In [618]: factor = qcut(data, [0, .25, .5, .75, 1.])

In [619]: data.groupby(factor).mean()
Out[619]:
[-3.469, -0.737]  -1.269581
(-0.737, 0.214]  -0.216269
(0.214, 1.0572]  0.680402
(1.0572, 3.0762]  1.629338
dtype: float64
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.

### 11.1 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:

```python
In [1055]: df = DataFrame(np.random.randn(10, 4))
In [1056]: df
Out[1056]:
   0    1       2        3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209  1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5  0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914

# break it into pieces
In [1057]: pieces = [df[:3], df[3:7], df[7:]]
In [1058]: concatenated = concat(pieces)
In [1059]: concatenated
Out[1059]:
   0   1    2     3
0  0.469112 -0.282863 -1.509059 -1.135632
1  1.212112 -0.173215  0.119209  1.044236
2 -0.861849 -2.104569 -0.494929  1.071804
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5  0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268
7 -0.370647 -1.157892 -1.344312  0.844885
8  1.075770 -0.109050  1.643563 -1.469388
9  0.357021 -0.674600 -1.776904 -0.968914
```
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”:

```python
concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
       keys=None, levels=None, names=None, verify_integrity=False)
```

- `objs`: list or dict 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)
- `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
- `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. If keys passed, specific levels to use for the resulting 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
- `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.

Without a little bit of context and example many of these arguments don’t make much sense. Let’s take 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:

```python
In [1060]: concatenated = concat(pieces, keys=[‘first’, ‘second’, ‘third’])
```

```python
In [1061]: concatenated
Out[1061]:
```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.469112</td>
<td>-0.282863</td>
<td>-1.509059</td>
</tr>
<tr>
<td>1</td>
<td>1.212112</td>
<td>-0.173215</td>
<td>0.119209</td>
</tr>
<tr>
<td>2</td>
<td>-0.861849</td>
<td>-2.104569</td>
<td>-0.494929</td>
</tr>
<tr>
<td>3</td>
<td>0.721555</td>
<td>-0.706771</td>
<td>-1.039575</td>
</tr>
<tr>
<td>4</td>
<td>-0.424972</td>
<td>0.567020</td>
<td>0.276232</td>
</tr>
<tr>
<td>5</td>
<td>-0.673690</td>
<td>0.113648</td>
<td>-1.478427</td>
</tr>
<tr>
<td>6</td>
<td>0.404705</td>
<td>0.577046</td>
<td>-1.715002</td>
</tr>
<tr>
<td>7</td>
<td>-0.370647</td>
<td>-1.157892</td>
<td>-1.344312</td>
</tr>
<tr>
<td>8</td>
<td>1.075770</td>
<td>-0.109050</td>
<td>1.643563</td>
</tr>
<tr>
<td>9</td>
<td>0.357021</td>
<td>-0.674600</td>
<td>-1.776904</td>
</tr>
</tbody>
</table>

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 do stuff like select out each chunk by key:
In [1062]: concatenated.ix[‘second’]
Out[1062]:
   0   1   2   3
3  0.721555 -0.706771 -1.039575  0.271860
4 -0.424972  0.567020  0.276232 -1.087401
5 -0.673690  0.113648 -1.478427  0.524988
6  0.404705  0.577046 -1.715002 -1.039268

It’s not a stretch to see how this can be very useful. More detail on this functionality below.

11.1.1 Set logic on the other axes

When gluing together multiple DataFrames (or Panels or...), for example, you have a choice of how to handle the other axes (other than the one being concatenated). This can be done in three ways:

- Take the (sorted) 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 (in the case of DataFrame) or indexes (in the case of Panel or future higher dimensional objects), i.e. the join_axes argument

Here is an example of each of these methods. First, the default join=’outer’ behavior:

In [1063]: from pandas.util.testing import rands

In [1064]: df = DataFrame(np.random.randn(10, 4), columns=[‘a’, ‘b’, ‘c’, ‘d’],
                   index=[rands(5) for _ in xrange(10)])

In [1065]: df
Out[1065]:
   a      b      c      d
AGx0N -1.294524  0.413738  0.276662 -0.472035
ML14H -0.013960 -0.362543 -0.006154 -0.923061
AkPA8  0.895717  0.805244 -1.206412  2.565646
oGYzQ  1.431256  1.340309 -1.170299 -0.226169
ZRFw   0.410835  0.813850  0.132003 -0.827317
HRxqN  0.759060 -2.211372  0.974466 -2.006747
HxpJj  1.413681  1.607920  1.024180  0.569605
J1MQR  0.875906 -2.211372  0.974466 -2.006747
sAcFy  0.410001 -0.078638  0.545952 -1.219217
UOsfn -1.226825  0.769804 -1.281247 -0.727707

In [1066]: concat([df.ix[7:], [‘a’, ‘b’]], df.ix[2:2, [‘c’]],
                   df.ix[-7:, [‘d’]], axis=1)

Out[1066]:
   a      b      c      d
AGx0N -1.294524  0.413738  NaN   NaN
AkPA8  0.895717  0.805244 -1.206412  NaN
HRxqN  0.759060 -2.211372  0.974466 -2.006747
HxpJj  1.413681  1.607920  1.024180  0.569605
J1MQR  NaN     NaN     0.974466 -2.006747
ML14H -0.013960 -0.362543  NaN   NaN
UOsfn  NaN     NaN     NaN   -0.727707
ZRFw   0.410835  0.813850  0.132003 -0.827317
Note that the row indexes have been unioned and sorted. Here is the same thing with `join='inner'`:

```
In [1067]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
           ......:  df.ix[-7:, ['d']]], axis=1, join='inner')
......:
Out[1067]:
```

Lastly, suppose we just wanted to reuse the exact index from the original DataFrame:

```
In [1068]: concat([df.ix[:7, ['a', 'b']], df.ix[2:-2, ['c']],
           ......:  df.ix[-7:, ['d']]], axis=1, join_axes=[df.index])
```

### 11.1.2 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 [1069]: s = Series(randn(10), index=np.arange(10))
In [1070]: s1 = s[:5] # note we’re slicing with labels here, so 5 is included
In [1071]: s2 = s[6:]
```

```
In [1072]: s1.append(s2)
```

In the case of DataFrame, the indexes must be disjoint but the columns do not need to be:
In [1073]: df = DataFrame(randn(6, 4), index=date_range('1/1/2000', periods=6),
    ...:       columns=['A', 'B', 'C', 'D'])
    
In [1074]: df1 = df.ix[:3]
In [1075]: df2 = df.ix[3:, :3]
In [1076]: df1
Out[1076]:
   A       B       C       D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
In [1077]: df2
Out[1077]:
   A       B       C
2000-01-04  0.690579  0.995761  2.396780
2000-01-05  3.357427 -0.317441 -1.236269
2000-01-06 -0.487602 -0.082240 -2.182937
In [1078]: df1.append(df2)
Out[1078]:
   A       B       C       D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
2000-01-04  0.690579  0.995761  2.396780   NaN
2000-01-05  3.357427 -0.317441 -1.236269   NaN
2000-01-06 -0.487602 -0.082240 -2.182937   NaN

append may take multiple objects to concatenate:
In [1079]: df1 = df.ix[:2]
In [1080]: df2 = df.ix[2:4]
In [1081]: df3 = df.ix[4:]
In [1082]: df1.append([df2,df3])
Out[1082]:
   A       B       C       D
2000-01-01  0.176444  0.403310 -0.154951  0.301624
2000-01-02 -2.179861 -1.369849 -0.954208  1.462696
2000-01-03 -1.743161 -0.826591 -0.345352  1.314232
2000-01-04  0.690579  0.995761  2.396780  0.014871
2000-01-05  3.357427 -0.317441 -1.236269  0.896171
2000-01-06 -0.487602 -0.082240 -2.182937  0.380396

Note: Unlike list.append method, which appends to the original list and returns nothing, append here does not modify df1 and returns its copy with df2 appended.
11.1.3 Ignoring indexes on the concatenation axis

For DataFrames which don’t have a meaningful index, you may wish to append them and ignore the fact that they may have overlapping indexes:

```python
In [1083]: df1 = DataFrame(randn(6, 4), columns=['A', 'B', 'C', 'D'])
In [1084]: df2 = DataFrame(randn(3, 4), columns=['A', 'B', 'C', 'D'])
In [1085]: df1
Out[1085]:
    A       B        C       D
0 0.084844 0.432390 1.519970 -0.493662
1 0.600178 0.274230 0.132885 -0.023688
2 2.410179 1.450520 0.206053 -0.251905
3 -2.213588 1.063327 1.266143 0.299368
4 -0.863838 0.408204 -1.048089 -0.025747
5 -0.988387 0.094055 1.262731 1.289997

In [1086]: df2
Out[1086]:
    A       B        C       D
0 0.082423 -0.055758 0.536580 -0.489682
1 0.369374 -0.034571 -2.484478 -0.281461
2 0.030711 0.109121 1.126203 -0.977349

To do this, use the ignore_index argument:

```python
In [1087]: concat([df1, df2], ignore_index=True)
Out[1087]:
    A       B        C       D
0 0.084844 0.432390 1.519970 -0.493662
1 0.600178 0.274230 0.132885 -0.023688
2 2.410179 1.450520 0.206053 -0.251905
3 -2.213588 1.063327 1.266143 0.299368
4 -0.863838 0.408204 -1.048089 -0.025747
5 -0.988387 0.094055 1.262731 1.289997
6 0.082423 -0.055758 0.536580 -0.489682
7 0.369374 -0.034571 -2.484478 -0.281461
8 0.030711 0.109121 1.126203 -0.977349

This is also a valid argument to DataFrame.append:

```python
In [1088]: df1.append(df2, ignore_index=True)
Out[1088]:
    A       B        C       D
0 0.084844 0.432390 1.519970 -0.493662
1 0.600178 0.274230 0.132885 -0.023688
2 2.410179 1.450520 0.206053 -0.251905
3 -2.213588 1.063327 1.266143 0.299368
4 -0.863838 0.408204 -1.048089 -0.025747
5 -0.988387 0.094055 1.262731 1.289997
6 0.082423 -0.055758 0.536580 -0.489682
7 0.369374 -0.034571 -2.484478 -0.281461
8 0.030711 0.109121 1.126203 -0.977349
11.1.4 More concatenating with group keys

Let's consider a variation on the first example presented:

In [1089]: df = DataFrame(np.random.randn(10, 4))

In [1090]: df
Out[1090]:
   0    1    2    3
0  1.47  0.50 -1.32  0.78
1 -1.07  0.44  2.35  0.58
2  0.22 -0.74  0.76  1.73
3 -0.96 -0.85 -1.34  1.85
4 -1.32  1.68 -1.72  0.89
5  0.22  0.90  1.17  0.52
6 -1.19 -1.07 -0.30 -0.86
7  0.59  0.74  0.47 -0.24
8  1.59  0.48  0.47 -1.46
9 -0.01 -0.28  0.65 -1.46

# break it into pieces
In [1091]: pieces = [df.ix[:, [0, 1]], df.ix[:, [2]], df.ix[:, [3]]]

In [1092]: result = concat(pieces, axis=1, keys=['one', 'two', 'three'])

In [1093]: result
Out[1093]:
   one   two   three
0  1.47  0.50 -1.32
1 -1.07  0.44  2.35
2  0.22 -0.74  0.76
3 -0.96 -0.85 -1.34
4 -1.32  1.68 -1.72
5  0.22  0.90  1.17
6 -1.19 -1.07 -0.30
7  0.59  0.74  0.47
8  1.59  0.48  0.47
9 -0.01 -0.28  0.65

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 [1094]: pieces = {'one': df.ix[:, [0, 1]], ....: 'two': df.ix[:, [2]], ....: 'three': df.ix[:, [3]]}

In [1095]: concat(pieces, axis=1)
Out[1095]:
   one   three   two
0  1.47  0.50 -1.32
1 -1.07  0.44  2.35
2  0.22 -0.74  0.76
3 -0.96 -0.85 -1.34
4 -1.32  1.68 -1.72
5  0.22  0.90  1.17
6 -1.19 -1.07 -0.30
7  0.59  0.74  0.47
8  1.59  0.48  0.47
9 -0.01 -0.28  0.65

11.1. Concatenating objects
In [1096]: `concat(pieces, keys=['three', 'two'])`
In [1096]:

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>three</td>
<td>0</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>NaN</td>
</tr>
<tr>
<td>two</td>
<td>0</td>
<td>-1.282782</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2.353925</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0.758527</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-1.340896</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>-1.717693</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>1.171216</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>-0.303421</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>0.384316</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.473424</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>0.650776</td>
</tr>
</tbody>
</table>

The MultiIndex created has levels that are constructed from the passed keys and the columns of the DataFrame pieces:

In [1097]: `result.columns.levels`
Out[1097]: `([Index(['one', 'two', 'three']), Int64Index([0, 1, 2, 3]), dtype=int64])`

If you wish to specify other levels (as will occasionally be the case), you can do so using the `levels` argument:

In [1098]: `result = concat(pieces, axis=1, keys=['one', 'two', 'three'], levels=[['three', 'two', 'one', 'zero']], names=['group_key'])`
   .......

In [1099]: `result`
Out[1099]:

<table>
<thead>
<tr>
<th>group_key</th>
<th>one</th>
<th>two</th>
<th>three</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.474071</td>
<td>-0.064034</td>
<td>-1.282782</td>
</tr>
<tr>
<td>1</td>
<td>-1.071357</td>
<td>0.441153</td>
<td>2.353925</td>
</tr>
<tr>
<td>2</td>
<td>0.221471</td>
<td>-0.744471</td>
<td>0.758527</td>
</tr>
<tr>
<td>3</td>
<td>-0.964980</td>
<td>-0.845696</td>
<td>-1.340896</td>
</tr>
<tr>
<td>4</td>
<td>-1.328865</td>
<td>1.682706</td>
<td>-1.717693</td>
</tr>
<tr>
<td>5</td>
<td>0.228440</td>
<td>0.901805</td>
<td>1.171216</td>
</tr>
<tr>
<td>6</td>
<td>-1.197071</td>
<td>-1.066969</td>
<td>-0.303421</td>
</tr>
<tr>
<td>7</td>
<td>0.306996</td>
<td>-0.028665</td>
<td>0.384316</td>
</tr>
<tr>
<td>8</td>
<td>1.588931</td>
<td>0.476720</td>
<td>0.473424</td>
</tr>
<tr>
<td>9</td>
<td>-0.014805</td>
<td>-0.284319</td>
<td>0.650776</td>
</tr>
</tbody>
</table>

In [1100]: `result.columns.levels`
Out[1100]: `([Index(['three', 'two', 'one', 'zero']), Int64Index([0, 1, 2, 3]), dtype=int64])`
Yes, this is fairly esoteric, but is actually necessary for implementing things like GroupBy where the order of a categorical variable is meaningful.

11.1.5 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 [1101]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])

In [1102]: df
Out[1102]:
   A         B         C         D
0 -1.137707 -0.891060 -0.693921  1.613616
1  0.464000  0.227371 -0.496922  0.306389
2 -2.290613 -1.134623 -1.561819 -0.260838
3  0.281957  1.523962 -0.902937  0.068159
4 -0.057873 -0.368204 -1.144073  0.861209
5  0.800193  0.782098 -1.069094 -1.099248
6  0.255269  0.009750  0.661084  0.379319
7 -0.008434  1.952541 -1.056652  0.533946

In [1103]: s = df.xs(3)

In [1104]: df.append(s, ignore_index=True)
Out[1104]:
   A         B         C         D
0 -1.137707 -0.891060 -0.693921  1.613616
1  0.464000  0.227371 -0.496922  0.306389
2 -2.290613 -1.134623 -1.561819 -0.260838
3  0.281957  1.523962 -0.902937  0.068159
4 -0.057873 -0.368204 -1.144073  0.861209
5  0.800193  0.782098 -1.069094 -1.099248
6  0.255269  0.009750  0.661084  0.379319
7 -0.008434  1.952541 -1.056652  0.533946
8  0.281957  1.523962 -0.902937  0.068159

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 [1105]: df = DataFrame(np.random.randn(5, 4), columns=['foo', 'bar', 'baz', 'qux'])

In [1106]: dicts = [{'foo': 1, 'bar': 2, 'baz': 3, 'peekaboo': 4},
               {'foo': 5, 'bar': 6, 'baz': 7, 'peekaboo': 8}]

In [1107]: result = df.append(dicts, ignore_index=True)

In [1108]: result
Out[1108]:
   bar    baz     foo  peekaboo     qux
0  0.040403 -0.507516 -1.226970   NaN  0.230096
1 -1.934370 -1.652499  0.394500   NaN  1.488753
2  0.576897  1.146000 -0.896484   NaN  1.487349

11.1. Concatenating objects
11.2 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 internal layout of the data in DataFrame.

pandas provides a single function, merge, as the entry point for all standard database join operations between DataFrame objects:

```python
merge(left, right, how='left', on=None, left_on=None, right_on=None, 
      left_index=False, right_index=False, sort=True, 
      suffixes=('_x', '_y'), copy=True)
```

Here’s a description of what each argument is for:

- **left**: A DataFrame object
- **right**: Another DataFrame object
- **on**: Columns (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 from the left DataFrame to use as keys. Can either be column names or arrays with length equal to the length of the DataFrame
- **right_on**: Columns from the right DataFrame to use as keys. Can either be column 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.

merge is a function in the pandas namespace, and it is also available as a DataFrame instance method, with the calling DataFrame being implicitly considered the left object in the join.

The related DataFrame.join method, uses merge internally for the index-on-index and index-on-column(s) joins, but joins on indexes by default rather than trying to join on common columns (the default behavior for merge). If you are joining on index, you may wish to use DataFrame.join to save yourself some typing.
11.2.1 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 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:

```python
In [1109]: left = DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
In [1110]: right = DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})
In [1111]: merge(left, right, on='key')
Out[1111]:
    key lval rval
0   foo   1    4
1   foo   1    5
2   foo   2    4
3   foo   2    5
```

Here is a more complicated example with multiple join keys:

```python
In [1114]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
    ....:                   'key2': ['one', 'two', 'one'],
    ....:                   'lval': [1, 2, 3]})
In [1115]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
    ....:                   'key2': ['one', 'one', 'one', 'two'],
    ....:                   'rval': [4, 5, 6, 7]})
In [1116]: merge(left, right, how='outer')
Out[1116]:
    key1 key2  lval rval
0   foo   one   1    4
1   foo   one   1    5
2   foo   one   2    4
3   foo   one   2    5
4   foo   two   4    7
5   bar   one   4    5
6   bar   one   5    6
7   bar   two   6    7
```

11.2. Database-style DataFrame joining/merging
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:

<table>
<thead>
<tr>
<th>Merge method</th>
<th>SQL Join Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>left</td>
<td>LEFT OUTER JOIN</td>
<td>Use keys from left frame only</td>
</tr>
<tr>
<td>right</td>
<td>RIGHT OUTER JOIN</td>
<td>Use keys from right frame only</td>
</tr>
<tr>
<td>outer</td>
<td>FULL OUTER JOIN</td>
<td>Use union of keys from both frames</td>
</tr>
<tr>
<td>inner</td>
<td>INNER JOIN</td>
<td>Use intersection of keys from both frames</td>
</tr>
</tbody>
</table>

### 11.2.2 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 [1118]: df = DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])
In [1119]: df1 = df.ix[1:, ['A', 'B']]
In [1120]: df2 = df.ix[:5, ['C', 'D']]
In [1121]: df1
Out[1121]:
   A    B
0 -2.461467 -1.553902
1  1.771740 -0.670027
2 -3.201750  0.792716
3 -0.747169 -0.309038
4  0.936527  1.255746
5  0.062297 -0.110388
6  0.077849  0.629498
7  0.077849  0.629498
```

```
In [1122]: df2
Out[1122]:
   C    D
0 0.377953  0.493672
1 2.015523 -1.833722
2 0.049307 -0.521493
3 0.146111  1.903247
4 0.393876  1.861468
5-2.655452  1.219492
```

```
In [1123]: df1.join(df2)
Out[1123]:
```

```
A B C D
1 -2.461467 -1.553902 2.015523 -1.833722
2 1.771740 -0.670027 0.049307 -0.521493
3 -3.201750 0.792716 0.146111 1.903247
4 -0.747169 -0.309038 0.393876 1.861468
5 0.936527 1.255746 -2.655452 1.219492
6 0.062297 -0.110388 NaN NaN
7 0.077849 0.629498 NaN NaN

In [1124]: df1.join(df2, how='outer')
Out[1124]:
A B C D
0 NaN NaN 0.377953 0.493672
1 -2.461467 -1.553902 2.015523 -1.833722
2 1.771740 -0.670027 0.049307 -0.521493
3 -3.201750 0.792716 0.146111 1.903247
4 -0.747169 -0.309038 0.393876 1.861468
5 0.936527 1.255746 -2.655452 1.219492
6 0.062297 -0.110388 NaN NaN
7 0.077849 0.629498 NaN NaN

In [1125]: df1.join(df2, how='inner')
Out[1125]:
A B C D
1 -2.461467 -1.553902 2.015523 -1.833722
2 1.771740 -0.670027 0.049307 -0.521493
3 -3.201750 0.792716 0.146111 1.903247
4 -0.747169 -0.309038 0.393876 1.861468
5 0.936527 1.255746 -2.655452 1.219492

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 [1126]: merge(df1, df2, left_index=True, right_index=True, how='outer')
Out[1126]:
A B C D
0 NaN NaN 0.377953 0.493672
1 -2.461467 -1.553902 2.015523 -1.833722
2 1.771740 -0.670027 0.049307 -0.521493
3 -3.201750 0.792716 0.146111 1.903247
4 -0.747169 -0.309038 0.393876 1.861468
5 0.936527 1.255746 -2.655452 1.219492
6 0.062297 -0.110388 NaN NaN
7 0.077849 0.629498 NaN NaN

11.2.3 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)
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:

11.2. Database-style DataFrame joining/merging
In [1127]: df['key'] = ['foo', 'bar'] * 4

In [1128]: to_join = DataFrame(randn(2, 2), index=['bar', 'foo'],
......:             columns=['j1', 'j2'])

In [1129]: df
Out[1129]:
   A     B     C     D   key
0 -0.308853 -0.681087 0.377953 0.493672    foo
1 -2.461467 -1.553902 2.015523 -1.833722    bar
2  1.771740 -0.670027 0.049307 -0.521493    foo
3 -3.201750  0.792716 0.146111  1.903247    bar
4  0.747169 -0.309038 0.393876  1.861468    foo
5  0.936527  1.255746 -2.655452  1.219492    bar
6  0.062297 -0.110388 -1.184357 -0.558081    foo
7  0.077849  0.629498 -1.035260 -0.438229    bar

In [1130]: to_join
Out[1130]:
   j1     j2
bar 0.503703  0.413086
foo-1.139050  0.660342

In [1131]: df.join(to_join, on='key')
Out[1131]:
   A     B     C     D   key   j1     j2
0 -0.308853 -0.681087 0.377953 0.493672     foo -1.139050  0.660342
1 -2.461467 -1.553902 2.015523 -1.833722     bar  0.503703  0.413086
2  1.771740 -0.670027 0.049307 -0.521493     foo -1.139050  0.660342
3 -3.201750  0.792716 0.146111  1.903247     bar  0.503703  0.413086
4  0.747169 -0.309038 0.393876  1.861468     foo -1.139050  0.660342
5  0.936527  1.255746 -2.655452  1.219492     bar  0.503703  0.413086
6  0.062297 -0.110388 -1.184357 -0.558081     foo -1.139050  0.660342
7  0.077849  0.629498 -1.035260 -0.438229     bar  0.503703  0.413086

In [1132]: merge(df, to_join, left_on='key', right_index=True,
   ....: how='left', sort=False)
   ....:
Out[1132]:
   A     B     C     D   key   j1     j2
0 -0.308853 -0.681087 0.377953 0.493672     foo -1.139050  0.660342
1 -2.461467 -1.553902 2.015523 -1.833722     bar  0.503703  0.413086
2  1.771740 -0.670027 0.049307 -0.521493     foo -1.139050  0.660342
3 -3.201750  0.792716 0.146111  1.903247     bar  0.503703  0.413086
4  0.747169 -0.309038 0.393876  1.861468     foo -1.139050  0.660342
5  0.936527  1.255746 -2.655452  1.219492     bar  0.503703  0.413086
6  0.062297 -0.110388 -1.184357 -0.558081     foo -1.139050  0.660342
7  0.077849  0.629498 -1.035260 -0.438229     bar  0.503703  0.413086

To join on multiple keys, the passed DataFrame must have a MultiIndex:

In [1133]: 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=['first', 'second'])
In [1134]: to_join = DataFrame(np.random.randn(10, 3), index=index, 
    columns=['j_one', 'j_two', 'j_three'])

# a little relevant example with NAs
In [1135]: key1 = ['bar', 'bar', 'bar', 'foo', 'foo', 'baz', 'baz', 'qux', 'qux', 'snap']

In [1136]: key2 = ['two', 'one', 'three', 'one', 'two', 'one', 'two', 'three', 'one']

In [1137]: data = np.random.randn(len(key1))

In [1138]: data = DataFrame({'key1': key1, 'key2': key2, 
    'data': data})

In [1139]: data
Out[1139]:
    key1 key2
data  
0    bar  two
1    bar  one
2    bar  three
3    foo  one
4    foo  two
5    baz  one
6    baz  two
7    qux  two
8    qux  three
9    snap one

In [1140]: to_join
Out[1140]:
    j_one  j_two  j_three
first  second
foo    one     0.464794 -0.309337 -0.649593
      two     0.683758 -0.643834  0.421287
      three   1.032814 -1.290493  0.787872
bar    one     1.515707 -0.276487 -0.223762
      two     1.397431  1.503874 -0.478905
baz    two    -0.135950 -0.730327 -0.033277
      three   0.281151 -1.298915  2.819487
qux    one    -0.851985 -1.106952 -0.937731
      two    -1.537770  0.555759 -0.277282
      three  -0.390201  1.207122  0.178690

Now this can be joined by passing the two key column names:

In [1141]: data.join(to_join, on=['key1', 'key2'])
Out[1141]:
    key1  key2  j_one  j_two  j_three
0    bar  two   1.397431 1.503874 -0.478905
1    bar  one   1.515707 -0.276487 -0.223762
2    bar  three  0.464794  0.643834  0.421287
3    foo  one    0.464794 -0.309337 -0.649593
4    foo  two    0.683758 -0.643834  0.421287
5    baz  one    NaN       NaN       NaN
6    baz  two    NaN       NaN       NaN
7    qux  one    NaN       NaN       NaN
8    qux  two  1.207122  0.178690   NaN
9    snap one  1.207122  0.178690   NaN

11.2. Database-style DataFrame joining/merging
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 [1142]: data.join(to_join, on=['key1', 'key2'], how='inner')
```

```
Out[1142]:

<table>
<thead>
<tr>
<th></th>
<th>key1</th>
<th>key2</th>
<th>j_one</th>
<th>j_two</th>
<th>j_three</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bar</td>
<td>two</td>
<td>1.397431</td>
<td>1.503874</td>
<td>-0.478905</td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
<td>one</td>
<td>1.515707</td>
<td>-0.276487</td>
<td>-0.223762</td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>one</td>
<td>0.464794</td>
<td>-0.309337</td>
<td>-0.649593</td>
</tr>
<tr>
<td>4</td>
<td>foo</td>
<td>two</td>
<td>0.683758</td>
<td>-0.643834</td>
<td>0.421287</td>
</tr>
<tr>
<td>6</td>
<td>baz</td>
<td>two</td>
<td>-0.135950</td>
<td>-0.730759</td>
<td>-2.277282</td>
</tr>
<tr>
<td>7</td>
<td>qux</td>
<td>two</td>
<td>-1.132896</td>
<td>0.555759</td>
<td>-2.277282</td>
</tr>
<tr>
<td>8</td>
<td>qux</td>
<td>three</td>
<td>-2.006481</td>
<td>0.178690</td>
<td></td>
</tr>
</tbody>
</table>

As you can see, this drops any rows where there was no match.

### 11.2.4 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 [1143]: left = DataFrame({'key': ['foo', 'foo'], 'value': [1, 2]})
In [1144]: right = DataFrame({'key': ['foo', 'foo'], 'value': [4, 5]})
In [1145]: merge(left, right, on='key', suffixes=['_left', '_right'])
```

```
Out[1145]:

<table>
<thead>
<tr>
<th></th>
<th>value_left</th>
<th>value_right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>foo</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>foo</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>foo</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>foo</td>
<td>5</td>
</tr>
</tbody>
</table>
```

DataFrame.join has lsuffix and rsuffix arguments which behave similarly.

### 11.2.5 Merging Ordered Data

New in v0.8.0 is the ordered_merge function for combining time series and other ordered data. In particular it has an optional fill_method keyword to fill/interpolate missing data:

```
In [1146]: A
Out[1146]:

<table>
<thead>
<tr>
<th></th>
<th>key</th>
<th>lvalue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>3</td>
</tr>
</tbody>
</table>
```
In [1147]: B
Out[1147]:
   key  rvalue
0   b      1
1   c      2
2   d      3

In [1148]: ordered_merge(A, B, fill_method='ffill', left_by='group')
Out[1148]:
   group  key  lvalue  rvalue
0      a    a   1    NaN
1      a    b   1      1
2      a    c   2      2
3      a    d   2      3
4      a    e   3      3
5      b    a   1    NaN
6      b    b   1      1
7      b    c   2      2
8      b    d   2      3
9      b    e   3      3

11.2.6 Joining multiple DataFrame or Panel objects

A list or tuple of DataFrames can also be passed to DataFrame.join to join them together on their indexes. The same is true for Panel.join.

In [1149]: df1 = df.ix[:, ['A', 'B']]
In [1150]: df2 = df.ix[:, ['C', 'D']]
In [1151]: df3 = df.ix[:, ['key']]
In [1152]: df1
Out[1152]:
   A     B
0 -0.308853 -0.681087
1 -2.461467 -1.553902
2  1.771740 -0.670027
3 -3.201750  0.792716
4 -0.747169 -0.309038
5  0.936527  1.255746
6  0.062297 -0.110388
7  0.077849  0.629498

In [1153]: df1.join([df2, df3])
Out[1153]:
   A     B   C     D  key
0 -0.308853 -0.681087 0.377953 0.493672 foo
1 -2.461467 -1.553902 2.015523 -1.833722 bar
2  1.771740 -0.670027 0.049307 -0.521493 foo
3 -3.201750  0.792716 0.146111  1.903247 bar
4 -0.747169 -0.309038 0.393876  1.861468 foo
5  0.936527  1.255746 -2.655452  1.219492 bar
6  0.062297 -0.110388 -1.184357 -0.558081 foo
7  0.077849  0.629498 -1.035260 -0.438229 bar

11.2. Database-style DataFrame joining/merging
11.2.7 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 [1154]: df1 = DataFrame([[nan, 3., 5.], [-4.6, np.nan, nan],
                                  [nan, 7., nan]])
In [1155]: df2 = DataFrame([[42.6, np.nan, -8.2], [-5., 1.6, 4]],
                                  index=[1, 2])

For this, use the combine_first method:

In [1156]: df1.combine_first(df2)
Out[1156]:
          0    1    2
0  NaN  3.0  5.0
1 -4.6  NaN -8.2
2 -5.0  1.6  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 [1157]: df1.update(df2)
In [1158]: df1
Out[1158]:
          0    1    2
0  NaN  3.0  5.0
1 -42.6  NaN -8.2
2 -5.0  1.6  4.0
RESHAPING AND PIVOT TABLES

12.1 Reshaping by pivoting DataFrame objects

Data is often stored in CSV files or databases in so-called “stacked” or “record” format:

```python
In [1225]: df
Out[1225]:
   date variable  value
0 2000-01-03  00:00:00  A    0.469112
1 2000-01-04  00:00:00  A  -0.282863
2 2000-01-05  00:00:00  A  -1.509059
3 2000-01-03  00:00:00  B  -1.135632
4 2000-01-04  00:00:00  B   1.212112
5 2000-01-05  00:00:00  B  -0.173215
6 2000-01-03  00:00:00  C   0.119209
7 2000-01-04  00:00:00  C  -1.044236
8 2000-01-05  00:00:00  C  -0.861849
9 2000-01-03  00:00:00  D  -2.104569
10 2000-01-04  00:00:00  D  -0.494929
11 2000-01-05  00:00:00  D   1.071804
```

For the curious here is how the above DataFrame was created:

```python
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 DataFrame(data, columns=['date', 'variable', 'value'])
df = unpivot(tm.makeTimeDataFrame())
```

To select out everything for variable `A` we could do:

```python
In [1226]: df[df['variable'] == 'A']
Out[1226]:
   date variable  value
0 2000-01-03  00:00:00  A    0.469112
1 2000-01-04  00:00:00  A  -0.282863
2 2000-01-05  00:00:00  A  -1.509059
```

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, use the `pivot` function:
In [1227]: df.pivot(index='date', columns='variable', values='value')
Out[1227]:
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 [1228]: df['value2'] = df['value'] * 2
In [1229]: pivoted = df.pivot('date', 'variable')
In [1230]: pivoted
Out[1230]:
           value     value2
variable    A     B     C     D    A     B     C
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569  0.938225 -2.271265  0.238417
2000-01-04  -0.282863  1.212112 -1.044236  -0.494929  -0.565727  2.424224 -2.088472
2000-01-05  -1.509059  -0.173215 -0.861849   1.071804  -3.018117  -0.346429 -1.723698

You of course can then select subsets from the pivoted DataFrame:

In [1231]: pivoted['value2']
Out[1231]:
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.

12.2 Reshaping by stacking and unstacking

Closely related to the pivot function are the related stack and unstack functions currently available on Series and DataFrame. These functions are designed to work together with MultiIndex objects (see the section on hierarchical indexing). Here are essentially what these functions 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 from 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.

The clearest way to explain is by example. Let’s take a prior example data set from the hierarchical indexing section:
In [1232]: tuples = zip(*[['bar', 'bar', 'baz', 'baz',
......:
......:   foo', 'foo', 'qux', 'qux'],
......:
......:   ['one', 'two', 'one', 'two'],
......:
......:   ['one', 'two', 'one', 'two']])

In [1233]: index = MultiIndex.from_tuples(tuples, names=['first', 'second'])

In [1234]: df = DataFrame(randn(8, 2), index=index, columns=['A', 'B'])

In [1235]: df2 = df[:4]

In [1236]: df2
Out[1236]:

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>two</td>
</tr>
<tr>
<td>baz</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>two</td>
</tr>
</tbody>
</table>

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 [1237]: stacked = df2.stack()

In [1238]: stacked
Out[1238]:

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>baz</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>dtype: float64</td>
<td></td>
</tr>
</tbody>
</table>

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 [1239]: stacked.unstack()
Out[1239]:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>bar one</td>
<td>0.721555</td>
</tr>
<tr>
<td></td>
<td>-0.706771</td>
</tr>
<tr>
<td></td>
<td>-1.039575</td>
</tr>
<tr>
<td></td>
<td>0.271860</td>
</tr>
<tr>
<td>baz one</td>
<td>-0.424972</td>
</tr>
<tr>
<td></td>
<td>0.567020</td>
</tr>
<tr>
<td></td>
<td>0.276232</td>
</tr>
<tr>
<td></td>
<td>-1.087401</td>
</tr>
</tbody>
</table>

In [1240]: stacked.unstack(1)
Out[1240]:

<table>
<thead>
<tr>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>one</td>
</tr>
<tr>
<td>two</td>
</tr>
</tbody>
</table>
If the indexes have names, you can use the level names instead of specifying the level numbers:

```
In [1242]: stacked.unstack('second')
Out[1242]:
second   one  two
         A  -1.039575  0.276232
         B   0.271860 -1.087401
```

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.

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 `sortlevel`, of course). Here is a more complex example:

```
In [1243]: columns = MultiIndex.from_tuples([(A, 'cat'), (B, 'dog'),
                                           (B, 'cat'), (A, 'dog')],
                                          names=['exp', 'animal'])

In [1244]: df = DataFrame(randn(8, 4), index=index, columns=columns)

In [1245]: df2 = df.ix[[0, 1, 2, 4, 5, 7]]

In [1246]: df2
Out[1246]:
exp   A    cat   B    dog
animal first second
bar   one   -0.370647 -1.157892 -1.344312  0.844885
      two    1.075770  -0.109050  1.643563 -1.469388
baz   one    0.357021  -0.674600 -1.776904 -0.968914
      two    0.895717   0.805244 -1.206412  2.565646
foo   one   -0.013960 -0.362543 -0.006154 -0.923061
      two    0.410835   0.813850  0.132003 -0.827317
qux   two    0.410835   0.813850  0.132003 -0.827317

As mentioned above, `stack` can be called with a `level` argument to select which level in the columns to stack:

```
In [1247]: df2.stack('exp')
Out[1247]:
animal   A    cat   B    dog
first second
bar   one   -0.370647 -1.157892 -1.344312  0.844885
      two    1.075770  -0.109050  1.643563 -1.469388
baz   one    0.357021  -0.674600 -1.776904 -0.968914
      two    0.895717   0.805244 -1.206412  2.565646
foo   one   -0.013960 -0.362543 -0.006154 -0.923061
      two    0.410835   0.813850  0.132003 -0.827317
qux   two    0.410835   0.813850  0.132003 -0.827317
```
<table>
<thead>
<tr>
<th>Animal</th>
<th>Exp</th>
<th>First</th>
<th>Second</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
</tr>
</tbody>
</table>

Unstacking when the columns are a MultiIndex is also careful about doing the right thing:

**In [1249]:** df[:3].unstack(0)
**Out[1249]:**
```
exp
animal
first bar baz
bar -0.370647 0.357021 -1.157892
baz 0.084885 0.084885 -1.157892
second
one -0.370647 0.357021 -1.157892 -0.6746 -1.344312 -1.776904 0.844885 -0.968914
two 1.075770 NaN -1.019050 0.643563 NaN -1.469388 NaN
```

**In [1250]:** df2.unstack(1)
**Out[1250]:**
```
exp
animal
second
one two one two one two one two
bar -0.370647 1.075770 -1.157892 -0.109050 -1.344312 1.643563 0.844885 -1.469388
baz 0.357021 NaN -0.674600 NaN -1.776904 NaN -0.968914 NaN
foo -0.013960 0.895717 -0.362543 0.805244 -0.006154 -1.206412 -0.923061 2.565646
qux NaN 0.410835 NaN 0.813850 NaN 0.132003 NaN -0.827317
```

### 12.2. Reshaping by stacking and unstacking

223
12.3 Reshaping by Melt

The `melt` function found in `pandas.core.reshape` is useful to massage a DataFrame into a format where one or more columns are identifier variables, while all other columns, considered measured variables, are “pivoted” to the row axis, leaving just two non-identifier columns, “variable” and “value”.

For instance,

```python
In [1251]: cheese = DataFrame({'first': ['John', 'Mary'],
   ....:   'last': ['Doe', 'Bo'],
   ....:   'height': [5.5, 6.0],
   ....:   'weight': [130, 150]})

In [1252]: cheese
Out[1252]:
   first  last  height  weight
0  John   Doe     5.5      130
1  Mary    Bo     6.0      150
```

```python
In [1253]: melt(cheese, id_vars=['first', 'last'])
Out[1253]:
   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
```

12.4 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.

```python
In [1254]: df
Out[1254]:
   exp   A   B   A
animal  cat  dog  cat  dog
first  second
bar   one -0.370647 -1.157892 -1.344312 0.844885
   two  1.075770 -0.109050  1.643563 -1.469388
baz   one  0.357021 -0.674600 -1.776904 -0.968914
   two -1.294524  0.413738  0.276662 -0.472035
foo   one -0.013960 -0.362543 -0.006154 -0.923061
   two  0.895717  0.805244 -1.206412  2.565646
qux   one  1.431256  1.340309 -1.170299 -0.226169
   two  0.410835  0.813850  0.132003 -0.827317
```

```python
In [1255]: df.stack().mean(1).unstack()
Out[1255]:
   animal  first  second
bar    one -0.857479 -0.156504
   two  1.359676 -0.789219
baz    one -0.709942 -0.821757
   two -0.508931 -0.029148
foo    one -0.010057 -0.642802
```
two -0.155347 1.685445
qux one 0.130479 0.557070
two 0.271419 -0.006733

# same result, another way
In [1256]: df.groupby(level=1, axis=1).mean()
Out[1256]:
animal     cat    dog
first second
bar   one  -0.857479 -0.156504
two   1.359666   -0.789219
baz   one  -0.709942 -0.821757
two   -0.508931 -0.029148
foo   one  -0.010057 -0.642802
two   -0.155347  1.685445
qux   one   0.130479   0.557070
two   0.271419  -0.006733

In [1257]: df.stack().groupby(level=1).mean()
Out[1257]:
exp   A   B
second
one -0.016301 -0.644049
two  0.110588  0.346200

In [1258]: df.mean().unstack(0)
Out[1258]:
exp   A   B
animal
  cat -0.311433 -0.431481
  dog -0.184544  0.133632

12.5 Pivot tables and cross-tabulations

The function pandas.pivot_table can be used to create spreadsheet-style pivot tables. It takes a number of arguments

- data: A DataFrame object
- values: a column or a list of columns to aggregate
- rows: list of columns to group by on the table rows
- cols: list of columns to group by on the table columns
- aggfunc: function to use for aggregation, defaulting to numpy.mean

Consider a data set like this:

In [1259]: df = 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))

In [1260]: df
Out[1260]:
We can produce pivot tables from this data very easily:

```
In [1261]: pivot_table(df, values='D', rows=['A', 'B'], cols=['C'])
Out[1261]:
     bar  foo
  A
one  -1.154627 -0.243234
  B   -1.320253 -0.633158
  C   -1.219217 -0.826591
three  -0.128534  0.835120
  B     NaN       -0.079051
  C     NaN       -1.077692
  two  -1.327977 NaN
     NaN       0.316495  0.005518

In [1262]: pivot_table(df, values='D', rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
Out[1262]:
     one  three  two
  A
  B -2.309255 -0.486468 -2.655954 NaN NaN -0.257067
  C  2.377724  0.754600 -1.665013 NaN NaN  1.676079

In [1263]: pivot_table(df, values=['D', 'E'], rows=['B'], cols=['A', 'C'], aggfunc=np.sum)
Out[1263]:
D          E
  A
  B -2.309255 -0.486468 -2.655954 NaN NaN -0.257067 0.316495 0.005518
  B -2.640506 -1.266315 NaN -0.158102 1.670241 NaN -1.077692 0.352360
```
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 [1264]: pivot_table(df, rows=['A', 'B'], cols=['C'])
Out[1264]:
   D   E
  C   bar   foo   bar   foo
A B
one A -1.154627 -0.243234  0.158248  0.002759
  B -1.320253 -0.633158 -0.538846  0.176180
  C  1.188862  0.377300  1.000985  1.120915
three A -1.327977  NaN   -0.338421  NaN
    B  NaN  -0.079051  0.699535
    C -0.832506  NaN  -0.843645  NaN
two A  NaN  -0.128534  0.433512
    B  0.835120  NaN   0.588783
    C  0.838040  NaN  -1.181568
```

You can render a nice output of the table omitting the missing values by calling `to_string` if you wish:

```
In [1265]: table = pivot_table(df, rows=['A', 'B'], cols=['C'])

In [1266]: print table.to_string(na_rep='')
   D   E
  C   bar   foo   bar   foo
A B
one A -1.154627 -0.243234  0.158248  0.002759
  B -1.320253 -0.633158 -0.538846  0.176180
  C  1.188862  0.377300  1.000985  1.120915
three A -1.327977  NaN   -0.338421  NaN
    B  NaN  -0.079051  0.699535
    C -0.832506  NaN  -0.843645  NaN
two A  NaN  -0.128534  0.433512
    B  0.835120  NaN   0.588783
    C  0.838040  NaN  -1.181568
```

Note that `pivot_table` is also available as an instance method on DataFrame.

### 12.5.1 Cross tabulations

Use the `crosstab` function 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

- **rows**: array-like, values to group by in the rows
- **cols**: 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)

Any Series passed will have their name attributes used unless row or column names for the cross-tabulation are specified

For example:

```python
In [1267]: foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny', 'one', 'two'
In [1268]: a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)
In [1269]: b = np.array([one, one, two, one, two, one], dtype=object)
In [1270]: c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=object)
In [1271]: crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])
Out[1271]:
      b     c
    one  dull  shiny
    dull  1    0    0
    shiny 0    0    1
    all  2    1    1    0
```

### 12.5.2 Adding margins (partial aggregates)

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:

```python
In [1272]: df.pivot_table(rows=['A', 'B'], cols='C', margins=True, aggfunc=np.std)
Out[1272]:
    D     E
   | bar  | foo  | All    | bar  | foo  | All
---|------|------|--------|------|------|------
   A |      |      |        |      |      |      
   B |      |      |        |      |      |      
   C |      |      |        |      |      |      
   | NaN  | NaN  | NaN    | NaN  | NaN  | NaN  
   | 0.917338 | 0.917338 | NaN | 0.418926 | 0.418926 
   | 1.660627 | 1.660627 | 0.744165 | NaN | 0.744165 
   | NaN | NaN | NaN | NaN | NaN | NaN 
   | 1.630183 | 1.630183 | NaN | 0.363548 | 0.363548 
   | 0.197065 | 0.197065 | 3.915454 | NaN | 3.915454 
   | NaN | NaN | NaN | NaN | NaN | NaN 
   | 0.413074 | 0.413074 | NaN | 0.794212 | 0.794212 
   | NaN | NaN | NaN | NaN | NaN | NaN 
   | 1.294620 | 0.824989 | 1.064412 | 1.403041 | 1.188419 | 1.248988 
```

### 12.6 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 [1273]: ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])

In [1274]: cut(ages, bins=3)
Out[1274]:
Categorical:
array([[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]],
      dtype=object)
Levels (3): Index([9.95, 26.667, 26.667, 43.333, 60], dtype=object)

If the `bins` keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges:

In [1275]: cut(ages, bins=[0, 18, 35, 70])
Out[1275]:
Categorical:
array([[0, 18],
       [0, 18],
       [0, 18],
       [0, 18],
       [18, 35],
       [18, 35],
       [18, 35],
       [35, 70]],
      dtype=object)
Levels (3): Index([0, 18, 35, 70], dtype=object)
pandas has proven very successful as a tool for working with time series data, especially in the financial data analysis space. With the 0.8 release, we have further improved the time series API in pandas by leaps and bounds. Using the new NumPy `datetime64` dtype, 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:

```python
# 72 hours starting with midnight Jan 1st, 2011
In [1299]: rng = date_range('1/1/2011', periods=72, freq='H')
```

```python
In [1300]: rng[:5]
Out[1300]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-01 04:00:00]
Length: 5, Freq: H, Timezone: None
```

Index pandas objects with dates:

```python
In [1301]: ts = Series(randn(len(rng)), index=rng)
```

```python
In [1302]: ts.head()
Out[1302]:
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, dttype: float64
```

Change frequency and fill gaps:

```python
# to 45 minute frequency and forward fill
In [1303]: converted = ts.asfreq('45Min', method='pad')
```
In [1304]: converted.head()
Out[1304]:
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:

# Daily means
In [1305]: ts.resample('D', how='mean')
Out[1305]:
2011-01-01 -0.319569
2011-01-02 -0.337703
2011-01-03  0.117258
Freq: D, dtype: float64

13.1 Time Stamps vs. Time Spans

Time-stamped data is the most basic type of timeseries data that associates values with points in time. For pandas objects it means using the points in time to create the index

In [1306]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]

In [1307]: ts = Series(np.random.randn(3), dates)

In [1308]: type(ts.index)
Out[1308]: pandas.tseries.index.DatetimeIndex

In [1309]: ts
Out[1309]:
2012-05-01 -0.410001
2012-05-02 -0.078638
2012-05-03  0.545952
dtype: float64

However, in many cases it is more natural to associate things like change variables with a time span instead.

For example:

In [1310]: periods = PeriodIndex([Period('2012-01'), Period('2012-02'),
                           Period('2012-03')])

In [1311]: ts = Series(np.random.randn(3), periods)

In [1312]: type(ts.index)
Out[1312]: pandas.tseries.period.PeriodIndex

In [1313]: ts
Out[1313]:
2012-01 -1.219217
2012-02 -1.226825
2012-03  0.769804
Freq: M, dtype: float64
Starting with 0.8, 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.

### 13.2 Generating Ranges of Timestamps

To generate an index with time stamps, you can use either the DatetimeIndex or Index constructor and pass in a list of datetime objects:

```
In [1314]: dates = [datetime(2012, 5, 1), datetime(2012, 5, 2), datetime(2012, 5, 3)]
In [1315]: index = DatetimeIndex(dates)
In [1316]: index # Note the frequency information
Out[1316]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-05-01 00:00:00, ..., 2012-05-03 00:00:00]
Length: 3, Freq: None, Timezone: None
```

Practically, 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 pandas functions `date_range` and `bdate_range` to create timestamp indexes.

```
In [1319]: index = date_range('2000-1-1', periods=1000, freq='M')
In [1320]: index
Out[1320]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-31 00:00:00, ..., 2083-04-30 00:00:00]
Length: 1000, Freq: M, Timezone: None
```

Convenience functions like `date_range` and `bdate_range` utilize a variety of frequency aliases. The default frequency for `date_range` is a calendar day while the default for `bdate_range` is a business day.

```
In [1323]: start = datetime(2011, 1, 1)
In [1324]: end = datetime(2012, 1, 1)
In [1325]: rng = date_range(start, end)
```
In [1326]: rng
Out[1326]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2012-01-01 00:00:00]
Length: 366, Freq: D, Timezone: None

In [1327]: rng = bdate_range(start, end)

In [1328]: rng
Out[1328]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-12-30 00:00:00]
Length: 260, Freq: B, Timezone: None

date_range and bdate_range makes it easy to generate a range of dates using various combinations of parameters like start, end, periods, and freq:

In [1329]: date_range(start, end, freq='BM')
Out[1329]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-12-30 00:00:00]
Length: 12, Freq: BM, Timezone: None

In [1330]: date_range(start, end, freq='W')
Out[1330]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-02 00:00:00, ..., 2012-01-01 00:00:00]
Length: 53, Freq: W-SUN, Timezone: None

In [1331]: bdate_range(end=end, periods=20)
Out[1331]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-12-05 00:00:00, ..., 2011-12-30 00:00:00]
Length: 20, Freq: B, Timezone: None

In [1332]: bdate_range(start=start, periods=20)
Out[1332]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-28 00:00:00]
Length: 20, Freq: B, Timezone: None

The start and end dates are strictly inclusive. So it will not generate any dates outside of those dates if specified.

13.2.1 DatetimeIndex

One of the main uses for DatetimeIndex is as an index for pandas objects. The DatetimeIndex class contains many timeseries 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 can be used like a regular index and offers all of its intelligent functionality like selection, slicing, etc.

**In [1333]:** rng = date_range(start, end, freq='BM')
**In [1334]:** ts = Series(randn(len(rng)), index=rng)

**In [1335]:** ts.index
**Out[1335]:**

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-12-30 00:00:00]
Length: 12, Freq: BM, Timezone: None
```

**In [1336]:** ts[:5].index
**Out[1336]:**

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-05-31 00:00:00]
Length: 5, Freq: BM, Timezone: None
```

**In [1337]:** ts[::2].index
**Out[1337]:**

```
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-11-30 00:00:00]
Length: 6, Freq: 2BM, Timezone: None
```

You can pass in dates and strings that parses to dates as indexing parameters:

**In [1338]:** ts['1/31/2011']
**Out[1338]:** -1.2812473076599531

**In [1339]:** ts[datetime(2011, 12, 25):]
**Out[1339]:**

```
2011-12-30  0.687738
Freq: BM, dtype: float64
```

**In [1340]:** ts['10/31/2011':'12/31/2011']
**Out[1340]:**

```
2011-10-31  0.149748
2011-11-30 -0.732339
2011-12-30  0.687738
Freq: BM, dtype: float64
```

A truncate convenience function is provided that is equivalent to slicing:

**In [1341]:** ts.truncate(before='10/31/2011', after='12/31/2011')
**Out[1341]:**

```
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 [1342]:** ts['2011']
**Out[1342]:**

```
2011-01-31 -1.281247
2011-02-28 -0.727707
2011-03-31 -0.121306
```

13.2. Generating Ranges of Timestamps
In [1343]: ts['2011-6']
Out[1343]:
2011-06-30  0.341734
Freq: BM, dtype: float64

Even complicated fancy indexing that breaks the DatetimeIndex’s frequency regularity will result in a DatetimeIndex (but frequency is lost):

In [1344]: ts[[0, 2, 6]].index
Out[1344]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-31 00:00:00, ..., 2011-07-29 00:00:00]
Length: 3, Freq: None, Timezone: None

DatetimeIndex objects has all the basic functionality of regular Index objects and a smorgasbord of advanced timeseries-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. So please be careful.

13.3 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 pandas DateOffset, which represents a regular frequency increment. Specific offset logic like “month”, “business day”, or “one hour” is represented in its various subclasses.
<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DateOffset</td>
<td>Generic offset class, defaults to 1 calendar day</td>
</tr>
<tr>
<td>BDay</td>
<td>business day (weekday)</td>
</tr>
<tr>
<td>Week</td>
<td>one week, optionally anchored on a day of the week</td>
</tr>
<tr>
<td>WeekOfMonth</td>
<td>the x-th day of the y-th week of each month</td>
</tr>
<tr>
<td>MonthEnd</td>
<td>calendar month end</td>
</tr>
<tr>
<td>MonthBegin</td>
<td>calendar month begin</td>
</tr>
<tr>
<td>BMonthEnd</td>
<td>business month end</td>
</tr>
<tr>
<td>BMonthBegin</td>
<td>business month begin</td>
</tr>
<tr>
<td>QuarterEnd</td>
<td>calendar quarter end</td>
</tr>
<tr>
<td>QuarterBegin</td>
<td>calendar quarter begin</td>
</tr>
<tr>
<td>BQuarterEnd</td>
<td>business quarter end</td>
</tr>
<tr>
<td>BQuarterBegin</td>
<td>business quarter begin</td>
</tr>
<tr>
<td>YearEnd</td>
<td>calendar year end</td>
</tr>
<tr>
<td>YearBegin</td>
<td>calendar year begin</td>
</tr>
<tr>
<td>BYearEnd</td>
<td>business year end</td>
</tr>
<tr>
<td>BYearBegin</td>
<td>business year begin</td>
</tr>
<tr>
<td>Hour</td>
<td>one hour</td>
</tr>
<tr>
<td>Minute</td>
<td>one minute</td>
</tr>
<tr>
<td>Second</td>
<td>one second</td>
</tr>
<tr>
<td>Milli</td>
<td>one millisecond</td>
</tr>
<tr>
<td>Micro</td>
<td>one microsecond</td>
</tr>
</tbody>
</table>

The basic `DateOffset` takes the same arguments as `dateutil.relativedelta`, which works like:

```
In [1345]: d = datetime(2008, 8, 18)
In [1346]: d + relativedelta(months=4, days=5)
Out[1346]: datetime.datetime(2008, 12, 23, 0, 0)
```

We could have done the same thing with `DateOffset`:

```
In [1347]: from pandas.tseries.offsets import *
In [1348]: d + DateOffset(months=4, days=5)
Out[1348]: datetime.datetime(2008, 12, 23, 0, 0)
```

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:

```python
class BDay(DateOffset):
    
    def apply(self, other):
        ...
```

```
In [1349]: d - 5 * BDay()
Out[1349]: datetime.datetime(2008, 8, 11, 0, 0)

In [1350]: d + BMonthEnd()
Out[1350]: datetime.datetime(2008, 8, 29, 0, 0)
```
The \texttt{rollforward} and \texttt{rollback} methods do exactly what you would expect:

\begin{Verbatim}
\textbf{In [1351]:} \texttt{d}  \\
\textbf{Out [1351]:} \texttt{datetime.datetime(2008, 8, 18, 0, 0)}

\textbf{In [1352]:} \texttt{offset = BMonthEnd()}

\textbf{In [1353]:} \texttt{offset.rollforward(d)}  \\
\textbf{Out [1353]:} \texttt{datetime.datetime(2008, 8, 29, 0, 0)}

\textbf{In [1354]:} \texttt{offset.rollback(d)}  \\
\textbf{Out [1354]:} \texttt{datetime.datetime(2008, 7, 31, 0, 0)}
\end{Verbatim}

It’s definitely worth exploring the \texttt{pandas.tseries.offsets} module and the various docstrings for the classes.

### 13.3.1 Parametric offsets

Some of the offsets can be “parameterized” when created to result in different behavior. For example, the \texttt{Week} offset for generating weekly data accepts a \texttt{weekday} parameter which results in the generated dates always lying on a particular day of the week:

\begin{Verbatim}
\textbf{In [1355]:} \texttt{d + Week()}  \\
\textbf{Out [1355]:} \texttt{datetime.datetime(2008, 8, 25, 0, 0)}

\textbf{In [1356]:} \texttt{d + Week(weekday=4)}  \\
\textbf{Out [1356]:} \texttt{datetime.datetime(2008, 8, 22, 0, 0)}

\textbf{In [1357]:} \texttt{(d + Week(weekday=4)).weekday()}  \\
\textbf{Out [1357]:} 4
\end{Verbatim}

Another example is parameterizing \texttt{YearEnd} with the specific ending month:

\begin{Verbatim}
\textbf{In [1358]:} \texttt{d + YearEnd()}  \\
\textbf{Out [1358]:} \texttt{datetime.datetime(2008, 12, 31, 0, 0)}

\textbf{In [1359]:} \texttt{d + YearEnd(month=6)}  \\
\textbf{Out [1359]:} \texttt{datetime.datetime(2009, 6, 30, 0, 0)}
\end{Verbatim}

### 13.3.2 Offset Aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as \textit{offset aliases} (referred to as \textit{time rules} prior to v0.8.0).
### 13.3.3 Combining Aliases

As we have seen previously, the alias and the offset instance are fungible in most functions:

```
In [1360]: date_range(start, periods=5, freq='B')
Out[1360]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-07 00:00:00]
Length: 5, Freq: B, Timezone: None
```

```
In [1361]: date_range(start, periods=5, freq=BDay())
Out[1361]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-03 00:00:00, ..., 2011-01-07 00:00:00]
Length: 5, Freq: B, Timezone: None
```

You can combine together day and intraday offsets:

```
In [1362]: date_range(start, periods=10, freq='2h20min')
Out[1362]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-01 21:00:00]
Length: 10, Freq: 140T, Timezone: None
```

```
In [1363]: date_range(start, periods=10, freq='1D10U')
Out[1363]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2011-01-01 00:00:00, ..., 2011-01-10 00:00:00.000090]
Length: 10, Freq: 86400000010U, Timezone: None
```
13.3.4 Anchored Offsets

For some frequencies you can specify an anchoring suffix:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W-SUN</td>
<td>weekly frequency (sundays). Same as ‘W’</td>
</tr>
<tr>
<td>W-MON</td>
<td>weekly frequency (mondays)</td>
</tr>
<tr>
<td>W-TUE</td>
<td>weekly frequency (tuesdays)</td>
</tr>
<tr>
<td>W-WED</td>
<td>weekly frequency (wednesdays)</td>
</tr>
<tr>
<td>W-THU</td>
<td>weekly frequency (thursdays)</td>
</tr>
<tr>
<td>W-FRI</td>
<td>weekly frequency (fridays)</td>
</tr>
<tr>
<td>W-SAT</td>
<td>weekly frequency (saturdays)</td>
</tr>
<tr>
<td>(B)Q(S)-DEC</td>
<td>quarterly frequency, year ends in December. Same as ‘Q’</td>
</tr>
<tr>
<td>(B)Q(S)-JAN</td>
<td>quarterly frequency, year ends in January</td>
</tr>
<tr>
<td>(B)Q(S)-FEB</td>
<td>quarterly frequency, year ends in February</td>
</tr>
<tr>
<td>(B)Q(S)-MAR</td>
<td>quarterly frequency, year ends in March</td>
</tr>
<tr>
<td>(B)Q(S)-APR</td>
<td>quarterly frequency, year ends in April</td>
</tr>
<tr>
<td>(B)Q(S)-MAY</td>
<td>quarterly frequency, year ends in May</td>
</tr>
<tr>
<td>(B)Q(S)-JUN</td>
<td>quarterly frequency, year ends in June</td>
</tr>
<tr>
<td>(B)Q(S)-JUL</td>
<td>quarterly frequency, year ends in July</td>
</tr>
<tr>
<td>(B)Q(S)-AUG</td>
<td>quarterly frequency, year ends in August</td>
</tr>
<tr>
<td>(B)Q(S)-SEP</td>
<td>quarterly frequency, year ends in September</td>
</tr>
<tr>
<td>(B)Q(S)-OCT</td>
<td>quarterly frequency, year ends in October</td>
</tr>
<tr>
<td>(B)Q(S)-NOV</td>
<td>quarterly frequency, year ends in November</td>
</tr>
<tr>
<td>(B)A(S)-DEC</td>
<td>annual frequency, anchored end of December. Same as ‘A’</td>
</tr>
<tr>
<td>(B)A(S)-JAN</td>
<td>annual frequency, anchored end of January</td>
</tr>
<tr>
<td>(B)A(S)-FEB</td>
<td>annual frequency, anchored end of February</td>
</tr>
<tr>
<td>(B)A(S)-MAR</td>
<td>annual frequency, anchored end of March</td>
</tr>
<tr>
<td>(B)A(S)-APR</td>
<td>annual frequency, anchored end of April</td>
</tr>
<tr>
<td>(B)A(S)-MAY</td>
<td>annual frequency, anchored end of May</td>
</tr>
<tr>
<td>(B)A(S)-JUN</td>
<td>annual frequency, anchored end of June</td>
</tr>
<tr>
<td>(B)A(S)-JUL</td>
<td>annual frequency, anchored end of July</td>
</tr>
<tr>
<td>(B)A(S)-AUG</td>
<td>annual frequency, anchored end of August</td>
</tr>
<tr>
<td>(B)A(S)-SEP</td>
<td>annual frequency, anchored end of September</td>
</tr>
<tr>
<td>(B)A(S)-OCT</td>
<td>annual frequency, anchored end of October</td>
</tr>
<tr>
<td>(B)A(S)-NOV</td>
<td>annual frequency, anchored end of November</td>
</tr>
</tbody>
</table>

These can be used as arguments to `date_range`, `bdate_range`, constructors for `DatetimeIndex`, as well as various other timeseries-related functions in pandas.

13.3.5 Legacy Aliases

Note that prior to v0.8.0, time rules had a slightly different look. Pandas will continue to support the legacy time rules for the time being but it is strongly recommended that you switch to using the new offset aliases.
As you can see, legacy quarterly and annual frequencies are business quarter and business year ends. Please also note the legacy time rule for milliseconds $ms$ versus the new offset alias for month start $MS$. This means that offset alias parsing is case sensitive.

## 13.4 Time series-related instance methods

### 13.4.1 Shifting / lagging

One may want to *shift* or *lag* the values in a TimeSeries back and forward in time. The method for this is *shift*, which is available on all of the pandas objects. In DataFrame, *shift* will currently only shift along the *index* and in Panel along the *major_axis*.

In [1364]: ts = ts[:5]

In [1365]: ts.shift(1)

Out[1365]:
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 a *offset alias*:

```
In [1366]: ts.shift(5, freq=datetools.bday)
Out[1366]:
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 [1367]: ts.shift(5, freq='BM')
Out[1367]:
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 *TimeSeries* objects also have a *tshift* convenience method that changes all the dates in the index by a specified number of offsets:

```
In [1368]: ts.tshift(5, freq='D')
Out[1368]:
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.

### 13.4.2 Frequency conversion

The primary function for changing frequencies is the *asfreq* function. For a *DatetimeIndex*, this is basically just a thin, but convenient wrapper around *reindex* which generates a *date_range* and calls *reindex*.

```
In [1369]: dr = date_range('1/1/2010', periods=3, freq=3 * datetools.bday)
In [1370]: ts = Series(randn(3), index=dr)

In [1371]: ts
Out[1371]:
2010-01-01   0.176444
2010-01-06   0.403310
2010-01-11  -0.154951
Freq: 3B, dtype: float64
```

```
In [1372]: ts.asfreq(BDay())
Out[1372]:
2010-01-01   0.176444
2010-01-04   NaN
2010-01-05   NaN
2010-01-06   0.403310
2010-01-07   NaN
2010-01-08   NaN
```
2010-01-11  -0.154951
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

```python
In [1373]: ts.asfreq(BDay(), method='pad')
Out[1373]:
2010-01-01  0.176444
2010-01-04  0.176444
2010-01-05  0.176444
2010-01-06  0.403310
2010-01-07  0.403310
2010-01-08  0.403310
2010-01-11  -0.154951
Freq: B, dtype: float64
```

### 13.4.3 Filling forward / backward

Related to `asfreq` and `reindex` is the `fillna` function documented in the *missing data section*.

### 13.5 Up- and downsampling

With 0.8, pandas introduces 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.

```python
In [1374]: rng = date_range('1/1/2012', periods=100, freq='S')
In [1375]: ts = Series(randint(0, 500, len(rng)), index=rng)
In [1376]: ts.resample('5Min', how='sum')
Out[1376]:
2012-01-01  25792
Freq: 5T, dtype: float64
```

The `resample` function is very flexible and allows you to specify many different parameters to control the frequency
conversion and resampling operation.

The `how` parameter can be a function name or numpy array function that takes an array and produces aggregated
values:

```python
In [1377]: ts.resample('5Min')  # default is mean
Out[1377]:
2012-01-01  257.92
Freq: 5T, dtype: float64
```

```python
In [1378]: ts.resample('5Min', how='ohlc')
Out[1378]:
open   high   low  close
2012-01-01  230   492    0   214
```

```python
In [1379]: ts.resample('5Min', how=np.max)
Out[1379]:
```

### 13.5. Up- and downsampling

243
Any function available via *dispatching* can be given to the `how` parameter by name, including `sum`, `mean`, `std`, `max`, `min`, `median`, `first`, `last`, `ohlc`.

For downsampling, `closed` can be set to ‘left’ or ‘right’ to specify which end of the interval is closed:

```python
In [1380]: ts.resample('5Min', closed='right')
Out[1380]:
2011-12-31 23:55:00    230.000000
2012-01-01 00:00:00    258.202020
Freq: 5T, dtype: float64

In [1381]: ts.resample('5Min', closed='left')
Out[1381]:
2012-01-01 257.92
Freq: 5T, dtype: float64
```

For upsampling, the `fill_method` and `limit` parameters can be specified to interpolate over the gaps that are created:

```python
# from secondly to every 250 milliseconds
In [1382]: ts[0:2].resample('250L')
Out[1382]:
2012-01-01 00:00:00 0.250000   NaN
2012-01-01 00:00:00 0.500000   NaN
2012-01-01 00:00:00 0.750000   NaN
2012-01-01 00:00:01 0.000000   202
2012-01-01 00:00:00 1.250000   NaN
Freq: 250L, dtype: float64

In [1383]: ts[0:2].resample('250L', fill_method='pad')
Out[1383]:
2012-01-01 00:00:00 0.250000   230
2012-01-01 00:00:00 0.500000   230
2012-01-01 00:00:00 0.750000   230
2012-01-01 00:00:01 0.000000   202
2012-01-01 00:00:01 1.250000   202
Freq: 250L, dtype: int64

In [1384]: ts[0:2].resample('250L', fill_method='pad', limit=2)
Out[1384]:
2012-01-01 00:00:00 0.250000   230
2012-01-01 00:00:00 0.500000   230
2012-01-01 00:00:00 0.750000   NaN
2012-01-01 00:00:01 0.000000   202
2012-01-01 00:00:01 1.250000   202
Freq: 250L, 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.

```python
In [1385]: ts.resample('5Min')  # by default label='right'
Out[1385]:
2012-01-01 257.92
Freq: 5T, dtype: float64

In [1386]: ts.resample('5Min', label='left')
```
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.

Note that 0.8 marks a watershed in the timeseries functionality in pandas. In previous versions, resampling had to be done using a combination of `date_range`, `groupby` with `asof`, and then calling an aggregation function on the grouped object. This was not nearly convenient or performant as the new pandas timeseries API.

### 13.6 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`.

#### 13.6.1 Period

A `Period` represents a span of time (e.g., a day, a month, a quarter, etc). It can be created using a frequency alias:

```
In [1388]: Period('2012', freq='A-DEC')
Out[1388]: Period('2012', 'A-DEC')
```

```
In [1389]: Period('2012-1-1', freq='D')
Out[1389]: Period('2012-01-01', 'D')
```

```
In [1390]: Period('2012-1-1 19:00', freq='H')
Out[1390]: Period('2012-01-01 19:00', 'H')
```

Unlike time stamped data, pandas does not support frequencies at multiples of DateOffsets (e.g., ‘3Min’) for periods. Adding and subtracting integers from periods shifts the period by its own frequency.

```
In [1391]: p = Period('2012', freq='A-DEC')
```

```
In [1392]: p + 1
Out[1392]: Period('2013', 'A-DEC')
```

```
In [1393]: p - 3
Out[1393]: Period('2009', 'A-DEC')
```

Taking the difference of `Period` instances with the same frequency will return the number of frequency units between them:

```
In [1394]: Period('2012', freq='A-DEC') - Period('2002', freq='A-DEC')
Out[1394]: 10
```
13.6.2 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 [1395]: prng = period_range('1/1/2011', '1/1/2012', freq='M')
```

```
In [1396]: prng
Out[1396]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2011-01, ..., 2012-01]
length: 13
```

The PeriodIndex constructor can also be used directly:

```
In [1397]: PeriodIndex(['2011-1', '2011-2', '2011-3'], freq='M')
Out[1397]:
<class 'pandas.tseries.period.PeriodIndex'>
freq: M
[2011-01, ..., 2011-03]
length: 3
```

Just like DatetimeIndex, a PeriodIndex can also be used to index pandas objects:

```
In [1398]: Series(randn(len(prng)), prng)
Out[1398]:
2011-01  0.301624
2011-02 -1.460489
2011-03  0.610679
2011-04  1.195856
2011-05 -0.008820
2011-06 -0.045729
2011-07 -1.051015
2011-08 -0.422924
2011-09 -0.028361
2011-10 -0.782386
2011-11  0.861980
2011-12  1.438604
2012-01 -0.525492
Freq: M, dtype: float64
```

13.6.3 Frequency Conversion and Resampling with PeriodIndex

The frequency of Periods and PeriodIndex can be converted via the asfreq method. Let's start with the fiscal year 2011, ending in December:

```
In [1399]: p = Period('2011', freq='A-DEC')
```

```
In [1400]: p
Out[1400]: 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 [1401]: p.asfreq('M', how='start')
Out[1401]: Period('2011-01', 'M')
```
In [1402]: p.asfreq('M', how='end')
Out[1402]: Period('2011-12', 'M')

The shorthands ‘s’ and ‘e’ are provided for convenience:

In [1403]: p.asfreq('M', 's')
Out[1403]: Period('2011-01', 'M')

In [1404]: p.asfreq('M', 'e')
Out[1404]: 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 [1405]: p = Period('2011-12', freq='M')
In [1406]: p.asfreq('A-NOV')
Out[1406]: 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 start and ends. Thus, first quarter of 2011 could start in 2010
or a few months into 2011. Via anchored frequencies, pandas works all quarterly frequencies Q-JAN through Q-DEC.

Q-DEC define regular calendar quarters:

In [1407]: p = Period('2012Q1', freq='Q-DEC')
In [1408]: p.asfreq('D', 's')
Out[1408]: Period('2012-01-01', 'D')
In [1409]: p.asfreq('D', 'e')
Out[1409]: Period('2012-03-31', 'D')

Q-MAR defines fiscal year end in March:

In [1410]: p = Period('2011Q4', freq='Q-MAR')
In [1411]: p.asfreq('D', 's')
Out[1411]: Period('2011-01-01', 'D')
In [1412]: p.asfreq('D', 'e')
Out[1412]: Period('2011-03-31', 'D')

13.7 Converting between Representations

Timestamped data can be converted to PeriodIndex-ed data using to_period and vice-versa using
to_timestamp:

In [1413]: rng = date_range('1/1/2012', periods=5, freq='M')
In [1414]: ts = Series(randn(len(rng)), index=rng)
In [1415]: ts
Out[1415]:
2012-01-31    -1.684469

13.7. Converting between Representations 247
In [1416]: ps = ts.to_period()

In [1417]: ps
Out[1417]:
2012-01 -1.684469
2012-02 0.550605
2012-03 0.091955
2012-04 0.891713
2012-05 0.807078
Freq: M, dtype: float64

In [1418]: ps.to_timestamp()
Out[1418]:
2012-01-01 -1.684469
2012-02-01 0.550605
2012-03-01 0.091955
2012-04-01 0.891713
2012-05-01 0.807078
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 [1419]: ps.to_timestamp('D', how='s')
Out[1419]:
2012-01-01 -1.684469
2012-02-01 0.550605
2012-03-01 0.091955
2012-04-01 0.891713
2012-05-01 0.807078
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 [1420]: prng = period_range('1990Q1', '2000Q4', freq='Q-NOV')

In [1421]: ts = Series(randn(len(prng)), prng)

In [1422]: ts.index = (prng.asfreq('M', 'e') + 1).asfreq('H', 's') + 9

In [1423]: ts.head()
Out[1423]:
1990-03-01 09:00 0.221441
1990-06-01 09:00 -0.113139
1990-09-01 09:00 -1.812900
1990-12-01 09:00 -0.053708
1991-03-01 09:00 -0.114574
Freq: H, dtype: float64
13.8 Time Zone Handling

Using `pytz`, pandas provides rich support for working with timestamps in different time zones. By default, pandas objects are time zone unaware:

```
In [1424]: rng = date_range('3/6/2012 00:00', periods=15, freq='D')

In [1425]: print(rng.tz)
None
```

To supply the time zone, you can use the `tz` keyword to `date_range` and other functions:

```
In [1426]: rng_utc = date_range('3/6/2012 00:00', periods=10, freq='D', tz='UTC')

In [1427]: print(rng_utc.tz)
UTC
```

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 [1428]: ts = Series(randn(len(rng)), rng)

In [1429]: ts_utc = ts.tz_localize('UTC')

In [1430]: ts_utc
Out[1430]:
2012-03-06 00:00:00+00:00  -0.114722
2012-03-07 00:00:00+00:00  0.168904
2012-03-08 00:00:00+00:00  -0.048048
2012-03-09 00:00:00+00:00  0.801196
2012-03-10 00:00:00+00:00  1.392071
2012-03-11 00:00:00+00:00  -0.048788
2012-03-12 00:00:00+00:00  -0.808838
2012-03-13 00:00:00+00:00  -1.003677
2012-03-14 00:00:00+00:00  -0.160766
2012-03-15 00:00:00+00:00  1.758853
2012-03-16 00:00:00+00:00  0.729195
2012-03-17 00:00:00+00:00  1.359732
2012-03-18 00:00:00+00:00  2.006296
2012-03-19 00:00:00+00:00  0.870210
2012-03-20 00:00:00+00:00  0.043464
Freq: D, dtype: float64
```

You can use the `tz_convert` method to convert pandas objects to convert tz-aware data to another time zone:

```
In [1431]: ts_utc.tz_convert('US/Eastern')
Out[1431]:
2012-03-05 19:00:00-05:00  -0.114722
2012-03-06 19:00:00-05:00  0.168904
2012-03-07 19:00:00-05:00  -0.048048
2012-03-08 19:00:00-05:00  0.801196
2012-03-09 19:00:00-05:00  1.392071
2012-03-10 19:00:00-05:00  -0.048788
2012-03-11 20:00:00-04:00  -0.808838
2012-03-12 20:00:00-04:00  -1.003677
2012-03-13 20:00:00-04:00  -0.160766
2012-03-14 20:00:00-04:00  1.758853
2012-03-15 20:00:00-04:00  0.729195
2012-03-16 20:00:00-04:00  1.359732
```

13.8. Time Zone Handling 249
### 2012-03-17 20:00:00-04:00 2.006296
### 2012-03-18 20:00:00-04:00 0.870210
### 2012-03-19 20:00:00-04:00 0.043464
Freq: D, dtype: float64

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:

```python
In [1432]: rng_eastern = rng_utc.tz_convert('US/Eastern')
In [1433]: rng_berlin = rng_utc.tz_convert('Europe/Berlin')
In [1434]: rng_eastern[5]
Out[1434]: <Timestamp: 2012-03-10 19:00:00-0500 EST, tz=US/Eastern>
In [1435]: rng_berlin[5]
Out[1435]: <Timestamp: 2012-03-11 01:00:00+0100 CET, tz=Europe/Berlin>
Out[1436]: True
```

Like Series, DataFrame, and DatetimeIndex, Timestamps can be converted to other time zones using `tz_convert`:

```python
In [1437]: rng_eastern[5]
Out[1437]: <Timestamp: 2012-03-10 19:00:00-0500 EST, tz=US/Eastern>
In [1438]: rng_berlin[5]
Out[1438]: <Timestamp: 2012-03-11 01:00:00+0100 CET, tz=Europe/Berlin>
In [1439]: rng_eastern[5].tz_convert('Europe/Berlin')
Out[1439]: <Timestamp: 2012-03-11 01:00:00+0100 CET, tz=Europe/Berlin>
```

Localization of Timestamps functions just like DatetimeIndex and TimeSeries:

```python
In [1440]: rng[5]
Out[1440]: <Timestamp: 2012-03-11 00:00:00>
In [1441]: rng[5].tz_localize('Asia/Shanghai')
Out[1441]: <Timestamp: 2012-03-11 00:00:00+0800 CST, tz=Asia/Shanghai>
```

Operations between TimeSeries in difficult time zones will yield UTC TimeSeries, aligning the data on the UTC timestamps:

```python
In [1442]: eastern = ts_utc.tz_convert('US/Eastern')
In [1443]: berlin = ts_utc.tz_convert('Europe/Berlin')
In [1444]: result = eastern + berlin
In [1445]: result
Out[1445]:
2012-03-06 00:00:00+00:00 -0.229443
2012-03-07 00:00:00+00:00 0.337809
2012-03-08 00:00:00+00:00 -0.096096
2012-03-09 00:00:00+00:00 1.602392
2012-03-10 00:00:00+00:00 2.784142
2012-03-11 00:00:00+00:00 -0.097575
2012-03-12 00:00:00+00:00 -1.617677
```
In [1446]: result.index
Out[1446]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2012-03-06 00:00:00, ..., 2012-03-20 00:00:00]
Length: 15, Freq: D, Timezone: UTC
PLOTTING WITH MATPLOTLIB

Note: We intend to build more plotting integration with matplotlib as time goes on.

We use the standard convention for referencing the matplotlib API:

```
In [1447]: import matplotlib.pyplot as plt
```

### 14.1 Basic plotting: plot

The `plot` method on Series and DataFrame is just a simple wrapper around `plt.plot`:

```
In [1448]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))
In [1449]: ts = ts.cumsum()
In [1450]: ts.plot()
```

Out[1450]: `<matplotlib.axes.AxesSubplot at 0xacbdc10>`
If the index consists of dates, it calls `gcf().autofmt_xdate()` to try to format the x-axis nicely as per above. The method takes a number of arguments for controlling the look of the plot:

```python
In [1451]: plt.figure(); ts.plot(style='k--', label='Series'); plt.legend()
Out[1451]: <matplotlib.legend.Legend at 0x108d1610>
```

On DataFrame, `plot` is a convenience to plot all of the columns with labels:

```python
In [1452]: df = DataFrame(randn(1000, 4), index=ts.index,
                  ....: columns=['A', 'B', 'C', 'D'])
      ....:

In [1453]: df = df.cumsum()

In [1454]: plt.figure(); df.plot(); plt.legend(loc='best')
Out[1454]: <matplotlib.legend.Legend at 0x108c4450>
```
You may set the `legend` argument to `False` to hide the legend, which is shown by default.

```
In [1455]: df.plot(legend=False)
Out[1455]: <matplotlib.axes.AxesSubplot at 0x1132f8d0>
```

Some other options are available, like plotting each Series on a different axis:

```
In [1456]: df.plot(subplots=True, figsize=(8, 8)); plt.legend(loc='best')
Out[1456]: <matplotlib.legend.Legend at 0x1132f8d0>
```
You may pass `logy` to get a log-scale Y axis.

```
In [1457]: plt.figure();
In [1457]: ts = Series(randn(1000), index=date_range('1/1/2000', periods=1000))

In [1458]: ts = np.exp(ts.cumsum())

In [1459]: ts.plot(logy=True)
Out[1459]: <matplotlib.axes.AxesSubplot at 0x124f7e10>
```
You can plot one column versus another using the x and y keywords in `DataFrame.plot`:

```python
In [1460]: plt.figure()
Out[1460]: <matplotlib.figure.Figure at 0x11e1ec50>

In [1461]: df3 = DataFrame(np.random.randn(1000, 2), columns=['B', 'C']).cumsum()

In [1462]: df3['A'] = Series(range(len(df)))

In [1463]: df3.plot(x='A', y='B')
Out[1463]: <matplotlib.axes.AxesSubplot at 0x12d13c90>
```
14.1.1 Plotting on a Secondary Y-axis

To plot data on a secondary y-axis, use the `secondary_y` keyword:

```python
In [1464]: plt.figure()
Out[1464]: <matplotlib.figure.Figure at 0x12d22290>

In [1465]: df.A.plot()
Out[1465]: <matplotlib.axes.AxesSubplot at 0x1296d150>

In [1466]: df.B.plot(secondary_y=True, style='g')
Out[1466]: <matplotlib.axes.AxesSubplot at 0x1296d150>
```

14.1.2 Selective Plotting on Secondary Y-axis

To plot some columns in a DataFrame, give the column names to the `secondary_y` keyword:

```python
In [1467]: plt.figure()
Out[1467]: <matplotlib.figure.Figure at 0x130fbe90>

In [1468]: df.plot(secondary_y=['A', 'B'])
Out[1468]: <matplotlib.axes.AxesSubplot at 0x137563d0>
```
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 [1469]: plt.figure()
Out[1469]: <matplotlib.figure.Figure at 0x1354e210>
```
```
In [1470]: df.plot(secondary_y=['A', 'B'], mark_right=False)
Out[1470]: <matplotlib.axes.AxesSubplot at 0x13573b50>
```
14.1.3 Suppressing tick resolution adjustment

Pandas includes automatically 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 labelling is performed:

```
In [1471]: plt.figure()
Out[1471]: <matplotlib.figure.Figure at 0x13573050>

In [1472]: df.A.plot()
Out[1472]: <matplotlib.axes.AxesSubplot at 0x1356be90>
```

Using the `x_compat` parameter, you can suppress this behavior:

```
In [1473]: plt.figure()
Out[1473]: <matplotlib.figure.Figure at 0x14954650>

In [1474]: df.A.plot(x_compat=True)
Out[1474]: <matplotlib.axes.AxesSubplot at 0x14077450>
```
If you have more than one plot that needs to be suppressed, the `use` method in `pandas.plot_params` can be used in a `with` statement:

```python
In [1475]: import pandas as pd

In [1476]: plt.figure()
Out[1476]: <matplotlib.figure.Figure at 0x1498e910>

In [1477]: with pd.plot_params.use('x_compat', True):
......:    df.A.plot(color='r')
......:    df.B.plot(color='g')
......:    df.C.plot(color='b')
......:
```

14.1. Basic plotting: `plot`
14.1.4 Targeting different subplots

You can pass an ax argument to Series.plot to plot on a particular axis:

In [1478]: fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 5))

In [1479]: df['A'].plot(ax=axes[0,0]); axes[0,0].set_title('A')
Out[1479]: <matplotlib.text.Text at 0x14c74dd0>

In [1480]: df['B'].plot(ax=axes[0,1]); axes[0,1].set_title('B')
Out[1480]: <matplotlib.text.Text at 0x14fb5490>

In [1481]: df['C'].plot(ax=axes[1,0]); axes[1,0].set_title('C')
Out[1481]: <matplotlib.text.Text at 0x1530a990>

In [1482]: df['D'].plot(ax=axes[1,1]); axes[1,1].set_title('D')
Out[1482]: <matplotlib.text.Text at 0x1532c850>

14.2 Other plotting features

14.2.1 Bar plots

For labeled, non-time series data, you may wish to produce a bar plot:

In [1483]: plt.figure();
In [1483]: df.ix[5].plot(kind='bar'); plt.axhline(0, color='k')
Out[1483]: <matplotlib.lines.Line2D at 0x15cc0310>
Calling a DataFrame’s `plot` method with `kind='bar'` produces a multiple bar plot:

```python
In [1484]: df2 = DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])
In [1485]: df2.plot(kind='bar');
```

To produce a stacked bar plot, pass `stacked=True`:
In [1485]: df2.plot(kind='bar', stacked=True);

To get horizontal bar plots, pass kind='barh':

In [1485]: df2.plot(kind='barh', stacked=True);
### 14.2.2 Histograms

For a DataFrame, `hist` plots the histograms of the columns on multiple subplots:

```python
In [1486]: plt.figure()
Out[1486]: <matplotlib.figure.Figure at 0x1675d2d0>

In [1487]: df.diff().hist(color='k', alpha=0.5, bins=50)
Out[1487]:
array([[Axes(0.125,0.552174;0.336957x0.347826),
        Axes(0.563043,0.552174;0.336957x0.347826) ],
        [Axes(0.125,0.1;0.336957x0.347826),
         Axes(0.563043,0.1;0.336957x0.347826) ]], dtype=object)
```
New since 0.10.0, the `by` keyword can be specified to plot grouped histograms:

```
In [1488]: data = Series(np.random.randn(1000))
In [1489]: data.hist(by=np.random.randint(0, 4, 1000))
Out[1489]:
array([[[Axes(0.1,0.6;0.347826x0.3), Axes(0.552174,0.6;0.347826x0.3)],
        [Axes(0.1,0.15;0.347826x0.3), Axes(0.552174,0.15;0.347826x0.3)]],
    dtype=object)
```

14.2.3 Box-Plotting

DataFrame has a boxplot method which allows you 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 [1490]: df = DataFrame(np.random.rand(10,5))
```
You can create a stratified boxplot using the `by` keyword argument to create groupings. For instance,

In [1492]: df = DataFrame(np.random.rand(10, 2), columns=['Col1', 'Col2'])


In [1494]: plt.figure();
In [1494]: bp = df.boxplot(by='X')
You can also pass a subset of columns to plot, as well as group by multiple columns:

```
In [1495]: df = DataFrame(np.random.rand(10, 3), columns=['Col1', 'Col2', 'Col3'])
In [1496]: df['X'] = Series(['A','A','A','A','A','B','B','B','B','B'])
In [1497]: df['Y'] = Series(['A','B','A','B','A','B','A','B','A','B'])
In [1498]: plt.figure();
In [1498]: bp = df.boxplot(column=['Col1','Col2'], by=['X','Y'])
```

14.2.4 Scatter plot matrix

*New in 0.7.3.* You can create a scatter plot matrix using the `scatter_matrix` method in `pandas.tools.plotting`:

```
In [1499]: from pandas.tools.plotting import scatter_matrix
In [1500]: df = DataFrame(np.random.randn(1000, 4), columns=['a', 'b', 'c', 'd'])
In [1501]: scatter_matrix(df, alpha=0.2, figsize=(8, 8), diagonal='kde')
```

Out[1501]:
```python
array([[Axes(0.125, 0.7; 0.19375x0.2), Axes(0.31875, 0.7; 0.19375x0.2)],
       [Axes(0.5125, 0.7; 0.19375x0.2), Axes(0.70625, 0.7; 0.19375x0.2)],
       [Axes(0.125, 0.5; 0.19375x0.2), Axes(0.31875, 0.5; 0.19375x0.2)],
       [Axes(0.5125, 0.5; 0.19375x0.2), Axes(0.70625, 0.5; 0.19375x0.2)],
       [Axes(0.125, 0.3; 0.19375x0.2), Axes(0.31875, 0.3; 0.19375x0.2)],
       [Axes(0.5125, 0.3; 0.19375x0.2), Axes(0.70625, 0.3; 0.19375x0.2)],
       [Axes(0.125, 0.1; 0.19375x0.2), Axes(0.31875, 0.1; 0.19375x0.2)],
       [Axes(0.5125, 0.1; 0.19375x0.2), Axes(0.70625, 0.1; 0.19375x0.2)]], dtype=object)
```
0.8.0 You can create density plots using the Series/DataFrame.plot and setting `kind='kde'`:

```
In [1502]: ser = Series(np.random.randn(1000))

In [1503]: ser.plot(kind='kde')
Out[1503]: <matplotlib.axes.AxesSubplot at 0x1aed1410>
```

14.2. Other plotting features
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. 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](link).

```python
In [1504]: from pandas import read_csv

In [1505]: from pandas.tools.plotting import andrews_curves

In [1506]: data = read_csv('data/iris.data')

In [1507]: plt.figure()
   Out[1507]: <matplotlib.figure.Figure at 0x1acf0990>

In [1508]: andrews_curves(data, 'Name')
   Out[1508]: <matplotlib.axes.AxesSubplot at 0x1acfd890>
```
14.2.6 Parallel Coordinates

Parallel coordinates is a plotting technique for plotting multivariate data. It 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 [1509]: from pandas import read_csv

In [1510]: from pandas.tools.plotting import parallel_coordinates

In [1511]: data = read_csv('data/iris.data')

In [1512]: plt.figure()
Out[1512]: <matplotlib.figure.Figure at 0x1b591550>

In [1513]: parallel_coordinates(data, 'Name')
Out[1513]: <matplotlib.axes.AxesSubplot at 0x1b9c6250>
14.2.7 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.

In [1514]: from pandas.tools.plotting import lag_plot

In [1515]: plt.figure()
Out[1515]: <matplotlib.figure.Figure at 0x1acf6590>

In [1516]: data = Series(0.1 * np.random.random(1000) +
                           0.9 * np.sin(np.linspace(-99 * np.pi, 99 * np.pi, num=1000)))

In [1517]: lag_plot(data)
Out[1517]: <matplotlib.axes.AxesSubplot at 0x1c20bfd0>
14.2.8 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.

```python
In [1518]: from pandas.tools.plotting import autocorrelation_plot
In [1519]: plt.figure()
Out[1519]: <matplotlib.figure.Figure at 0x1c206390>
In [1520]: data = Series(0.7 * np.random.random(1000) + 0.3 * np.sin(np.linspace(-9 * np.pi, 9 * np.pi, num=1000)))
In [1521]: autocorrelation_plot(data)
Out[1521]: <matplotlib.axes.AxesSubplot at 0x1c09d850>
```

14.2. Other plotting features
14.2.9 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 [1522]: from pandas.tools.plotting import bootstrap_plot

In [1523]: data = Series(np.random.random(1000))

In [1524]: bootstrap_plot(data, size=50, samples=500, color='grey')

Out[1524]: <matplotlib.figure.Figure at 0x1c0958d0>
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 it will be colored differently.

Note: The “Iris” dataset is available here.

In [1525]: from pandas import read_csv

In [1526]: from pandas.tools.plotting import radviz
In [1527]: data = read_csv('data/iris.data')

In [1528]: plt.figure()
Out[1528]: <matplotlib.figure.Figure at 0x1c20bad0>

In [1529]: radviz(data, 'Name')
Out[1529]: <matplotlib.axes.AxesSubplot at 0x1d097c10>
15.1 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. They can take a number of arguments:

- `filepath_or_buffer`: Either a string path to a file, or any object with a read method (such as an open file or StringIO).
- `sep` or `delimiter`: A delimiter / separator to split fields on. `read_csv` is capable of inferring the delimiter automatically in some cases by “sniffing.” The separator may be specified as a regular expression; for instance you may use `\s*` to indicate a pipe plus arbitrary whitespace.
- `delim_whitespace`: Parse whitespace-delimited (spaces or tabs) file (much faster than using a regular expression)
- `compression`: decompress ‘gzip’ and ‘bz2’ formats on the fly.
- `dialect`: string or `csv.Dialect` instance to expose more ways to specify the file format
- `dtype`: A data type name or a dict of column name to data type. If not specified, data types will be inferred.
- `header`: row number to use as the column names, and the start of the data. Defaults to 0 if no names passed, otherwise None. Explicitly pass `header=0` to be able to replace existing names.
- `skiprows`: A collection of numbers for rows in the file to skip. Can also be an integer to skip the first n rows
- `index_col`: column number, column name, or list of column numbers/names, to use as the index (row labels) of the resulting DataFrame. By default, it will number the rows without using any column, unless there is one more data column than there are headers, in which case the first column is taken as the index.
- `names`: List of column names to use as column names. To replace header existing in file, explicitly pass `header=0`.
- `na_values`: optional list of strings to recognize as NaN (missing values), either in addition to or in lieu of the default set.
- `true_values`: list of strings to recognize as `True`
- `false_values`: list of strings to recognize as `False`
- `keep_default_na`: whether to include the default set of missing values in addition to the ones specified in na_values
- `parse_dates`: if True then index will be parsed as dates (False by default). You can specify more complicated options to parse a subset of columns or a combination of columns into a single date column (list of ints or names, list of lists, or dict) [1, 2, 3] -> try parsing columns 1, 2, 3 each as a separate date column [[1, 3]] -> combine
columns 1 and 3 and parse as a single date column {'foo': [1, 3]} -> parse columns 1, 3 as date and call result 'foo'

- keep_date_col: if True, then date component columns passed into parse_dates will be retained in the output (False by default).
- date_parser: function to use to parse strings into datetime objects. If parse_dates is True, it defaults to the very robust dateutil.parser. Specifying this implicitly sets parse_dates as True. You can also use functions from community supported date converters from date_converters.py
- dayfirst: if True then uses the DD/MM international/European date format (This is False by default)
- thousands: specifies the thousands separator. If not None, then parser will try to look for it in the output and parse relevant data to integers. Because it has to essentially scan through the data again, this causes a significant performance hit so only use if necessary.
- comment: denotes the start of a comment and ignores the rest of the line. Currently line commenting is not supported.
- nrows: Number of rows to read out of the file. Useful to only read a small portion of a large file
- iterator: If True, return a TextParser to enable reading a file into memory piece by piece
- chunksize: An number of rows to be used to “chunk” a file into pieces. Will cause an TextParser object to be returned. More on this below in the section on iterating and chunking
- skip_footer: number of lines to skip at bottom of file (default 0)
- converters: a dictionary of functions for converting values in certain columns, where keys are either integers or column labels
- encoding: a string representing the encoding to use for decoding unicode data, e.g. ‘utf-8’ or ’latin-1’.
- verbose: show number of NA values inserted in non-numeric columns
- squeeze: if True then output with only one column is turned into Series

Consider a typical CSV file containing, in this case, some time series data:

```python
In [830]: print open('foo.csv').read()
date,A,B,C
20090101,a,1,2
20090102,b,3,4
20090103,c,4,5
```

The default for read_csv is to create a DataFrame with simple numbered rows:

```python
In [831]: pd.read_csv('foo.csv')
Out[831]:
date  A  B  C
0  20090101  a  1  2
1  20090102  b  3  4
2  20090103  c  4  5
```

In the case of indexed data, you can pass the column number or column name you wish to use as the index:

```python
In [832]: pd.read_csv('foo.csv', index_col=0)
Out[832]:
   A  B  C
date
20090101  a  1  2
20090102  b  3  4
20090103  c  4  5
```

Chapter 15. IO Tools (Text, CSV, HDF5, ...
In [833]: `pd.read_csv('foo.csv', index_col='date')`
Out[833]:
   A  B  C
date
20090101 a 1 2
20090102 b 3 4
20090103 c 4 5

You can also use a list of columns to create a hierarchical index:

In [834]: `pd.read_csv('foo.csv', index_col=[0, 'A'])`
Out[834]:
      B  C
date       A
20090101 a 1 2
20090102 b 3 4
20090103 c 4 5

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 [835]: `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 [836]: `dia = csv.excel()`
In [837]: `dia.quoting = csv.QUOTE_NONE`
In [838]: `pd.read_csv(StringIO(data), dialect=dia)`
Out[838]:
      label1 label2 label3
index1      "a  c  e
index2       b  d  f

All of the dialect options can be specified separately by keyword arguments:

In [839]: `data = 'a,b,c~1,2,3~4,5,6'`
In [840]: `pd.read_csv(StringIO(data), lineterminator='~')`
Out[840]:
   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 [841]: `data = 'a, b, c\n1, 2, 3\n4, 5, 6'`
In [842]: `print data`
a, b, c
1, 2, 3
4, 5, 6
The parsers make every attempt to “do the right thing” and not be very fragile. Type inference is a pretty big deal. So if a column can be coerced to integer dtype without altering the contents, it will do so. Any non-numeric columns will come through as object dtype as with the rest of pandas objects.

### 15.1.1 Specifying column data types

Starting with v0.10, you can indicate the data type for the whole DataFrame or individual columns:

```python
In [844]: data = 'a,b,c
1,2,3
4,5,6
7,8,9'

In [845]: print data
a,b,c
1,2,3
4,5,6
7,8,9

In [846]: df = pd.read_csv(StringIO(data), dtype=object)

In [847]: df
Out[847]:
   a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

In [848]: df['a'][0]
Out[848]: '1'

In [849]: df = pd.read_csv(StringIO(data), dtype={'b': object, 'c': np.float64})

In [850]: df.dtypes
Out[850]:
   a     int64
   b    object
   c  float64
dtype: object
```

### 15.1.2 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:

```python
In [851]: from StringIO import StringIO

In [852]: data = 'a,b,c
1,2,3
4,5,6
7,8,9'

In [853]: print data
a,b,c
1,2,3
4,5,6
```

280 Chapter 15. IO Tools (Text, CSV, HDF5, ...
In [854]: pd.read_csv(StringIO(data))
Out[854]:
     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 [855]: print data
   a,b,c
   1,2,3
   4,5,6
   7,8,9

In [856]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=0)
Out[856]:
    foo  bar  baz
   0  1  2  3
   1  4  5  6
   2  7  8  9

In [857]: pd.read_csv(StringIO(data), names=['foo', 'bar', 'baz'], header=None)
Out[857]:
     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 [858]: data = 'skip this skip it
   
a,b,c
   1,2,3
   4,5,6
   7,8,9'

In [859]: pd.read_csv(StringIO(data), header=1)
Out[859]:
     a  b  c
0  1  2  3
1  4  5  6
2  7  8  9

15.1.3 Filtering columns (usecols)

The usecols argument allows you to select any subset of the columns in a file, either using the column names or position numbers:

In [860]: data = 'a,b,c,d
   
   n1,2,3,foo
   n4,5,6,bar
   n7,8,9,baz'

In [861]: pd.read_csv(StringIO(data))
Out[861]:
     a  b  c  d
0  1  2  3  foo
1  4  5  6  bar
2  7  8  9  baz
In [862]: pd.read_csv(StringIO(data), usecols=['b', 'd'])
Out[862]:
   b  d
0  2  foo
1  5  bar
2  8  baz

In [863]: pd.read_csv(StringIO(data), usecols=[0, 2, 3])
Out[863]:
   a  c  d
0  1  3  foo
1  4  6  bar
2  7  9  baz

15.1.4 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 [864]: data = 'word,length
Träumen,7
Grüße,5'
In [865]: df = pd.read_csv(StringIO(data), encoding='latin-1')
In [866]: df['word'][1]
Out[866]: u'Grüß'

Some formats which encode all characters as multiple bytes, like UTF-16, won’t parse correctly at all without specifying the encoding.

15.1.5 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 [868]: data = 'a,b,c
4,apple,bat,5.7
8,orange,cow,10'
In [869]: pd.read_csv(StringIO(data))
Out[869]:
   a  b  c
4 apple bat  5.7
8 orange cow  10.0

In [870]: data = 'index,a,b,c
4,apple,bat,5.7
8,orange,cow,10'
In [871]: pd.read_csv(StringIO(data), index_col=0)
Out[871]:
   a  b  c
index
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`:

```python
In [872]: data = 'a,b,c
4,apple,bat,
8,orange,cow,'

In [873]:
print(data)
a,b,c
4,apple,bat,
8,orange,cow,

In [874]: pd.read_csv(StringIO(data))
Out[874]:
a b c
4 apple bat NaN
8 orange cow NaN

In [875]: pd.read_csv(StringIO(data), index_col=False)
Out[875]:
a b c
0 4 apple bat
1 8 orange cow
```

### 15.1.6 Specifying Date Columns

To better facilitate working with datetime data, `read_csv()` and `read_table()` uses 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`:

```python
# Use a column as an index, and parse it as dates.
In [876]: df = pd.read_csv('foo.csv', index_col=0, parse_dates=True)

In [877]: df
Out[877]:
A B C
date
2009-01-01 a 1 2
2009-01-02 b 3 4
2009-01-03 c 4 5

# These are python datetime objects
In [878]: df.index
Out[878]:<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01 00:00:00, ..., 2009-01-03 00:00:00]
Length: 3, Freq: None, Timezone: 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 [879]: 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 [880]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]])

In [881]: df
Out[881]:
   1_2  1_3   0  4
0  0   0   KORD 0.81
1  1   1   KORD 0.01
2  2   2   KORD -0.59
3  3   3   KORD -0.99
4  4   4   KORD -0.59
5  5   5   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 [882]: df = pd.read_csv('tmp.csv', header=None, parse_dates=[[1, 2], [1, 3]], keep_date_col=True)

In [883]: df
Out[883]:
   1_2  1_3   0  1  2  3  4
0  0   0   KORD 19990127 0.81
1  1   1   KORD 19990127 0.01
2  2   2   KORD 19990127 -0.59
3  3   3   KORD 19990127 -0.99
4  4   4   KORD 19990127 -0.59
5  5   5   KORD 19990127 -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 [884]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [885]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec)

In [886]: df
Out[886]:
   nominal  actual   0  4
0  1999-01-27 19:00:00 1999-01-27 18:56:00 0.81
1  1999-01-27 20:00:00 1999-01-27 19:56:00 0.01
2  1999-01-27 21:00:00 1999-01-27 20:56:00 -0.59
3  1999-01-27 21:00:00 1999-01-27 21:18:00 -0.99
4  1999-01-27 22:00:00 1999-01-27 21:56:00 -0.59
5  1999-01-27 23:00:00 1999-01-27 22:56:00 -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 [887]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}

In [888]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
.....:
                     index_col=0) # index is the nominal column

In [889]: df
Out[889]:

       actual
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: 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 then as an index on the resulting frame.

15.1.7 Date Parsing Functions

Finally, the parser allows you can specify a custom date_parser function to take full advantage of the flexiblity of the date parsing API:

In [890]: import pandas.io.date_converters as conv

In [891]: df = pd.read_csv('tmp.csv', header=None, parse_dates=date_spec,
.....:
                     date_parser=conv.parse_date_time)

In [892]: df
Out[892]:

       nominal    actual
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

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.

15.1.8 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 [893]: `print` open('tmp.csv').read()
date, value, cat
1/6/2000, 5, a
2/6/2000, 10, b
3/6/2000, 15, c

In [894]: `pd.read_csv('tmp.csv', parse_dates=[0])`
Out[894]:
<table>
<thead>
<tr>
<th>date</th>
<th>value</th>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-06</td>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>2000-02-06</td>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>2000-03-06</td>
<td>15</td>
<td>c</td>
</tr>
</tbody>
</table>

In [895]: `pd.read_csv('tmp.csv', dayfirst=True, parse_dates=[0])`
Out[895]:
<table>
<thead>
<tr>
<th>date</th>
<th>value</th>
<th>cat</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-06-01</td>
<td>5</td>
<td>a</td>
</tr>
<tr>
<td>2000-06-02</td>
<td>10</td>
<td>b</td>
</tr>
<tr>
<td>2000-06-03</td>
<td>15</td>
<td>c</td>
</tr>
</tbody>
</table>

15.1.9 Thousand Separators

For large integers that have been written with a thousands separator, you can set the `thousands` keyword to `True` so that integers will be parsed correctly:

By default, integers with a thousands separator will be parsed as strings

In [896]: `print` open('tmp.csv').read()
ID| level| category
Patient1| 123,000| x
Patient2| 23,000| y
Patient3| 1,234,018| z

In [897]: df = `pd.read_csv('tmp.csv', sep='|')`

In [898]: df
Out[898]:
<table>
<thead>
<tr>
<th>ID</th>
<th>level</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patient1</td>
<td>123,000</td>
<td>x</td>
</tr>
<tr>
<td>Patient2</td>
<td>23,000</td>
<td>y</td>
</tr>
<tr>
<td>Patient3</td>
<td>1,234,018</td>
<td>z</td>
</tr>
</tbody>
</table>

In [899]: df.level.dtype
Out[899]: `dtype('object')`

The `thousands` keyword allows integers to be parsed correctly

In [900]: `print` open('tmp.csv').read()
ID| level| category
Patient1| 123,000| x
Patient2| 23,000| y
Patient3| 1,234,018| z

In [901]: df = `pd.read_csv('tmp.csv', sep='|', thousands='',)`

In [902]: df
Out[902]:
15.1.10 Comments

Sometimes comments or meta data may be included in a file:

```python
In [904]: 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 parse includes the comments in the output:

```python
In [905]: df = pd.read_csv('tmp.csv')
In [906]: df
Out[906]:
    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:

```python
In [907]: df = pd.read_csv('tmp.csv', comment='#')
In [908]: df
Out[908]:
    ID    level category
0  Patient1  123000      x
1  Patient2  23000       y
2  Patient3  1234018     z
```

15.1.11 Returning Series

Using the `squeeze` keyword, the parser will return output with a single column as a `Series`:

```python
In [909]: print open('tmp.csv').read()
level
Patient1,123000
Patient2,23000
Patient3,1234018
```

```python
In [910]: output = pd.read_csv('tmp.csv', squeeze=True)
In [911]: output
Out[911]:
        Patient1  123000
Patient1            123000
Patient2            23000
Patient3            1234018
```
Patient3 1234018
Name: level, dtype: int64

In [912]: type(output)
Out[912]: pandas.core.series.Series

15.1.12 Boolean values

The common values True, False, TRUE, and FALSE are all recognized as boolean. Sometime you would want to recognize some other values as being boolean. To do this use the true_values and false_values options:

In [913]: data= 'a,b,c
1,Yes,2
3,No,4'

In [914]: print data  
a,b,c
1,Yes,2
3,No,4

In [915]: pd.read_csv(StringIO(data))
Out[915]:
   a   b  c
0  1 Yes 2
1  3  No 4

In [916]: pd.read_csv(StringIO(data), true_values=['Yes'], false_values=['No'])
Out[916]:
   a   b   c
0  1 True 2
1  3 False 4

15.1.13 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 will cause an error by default:

In [27]: data = 'a,b,c
1,2,3
4,5,6,7
8,9,10'

In [28]: pd.read_csv(StringIO(data))
---------------------------------------------------------------------------
CParserError Traceback (most recent call last)
CParserError: 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
15.1.14 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:

```python
In [917]: data = 'a,b

"hello, \"Bob\"", nice to see you",5'
In [918]: print data
a,b
"hello, "Bob\", nice to see you",5
In [919]: pd.read_csv(StringIO(data), escapechar='\')
Out[919]:
   a     b
0 hello, "Bob", nice to see you 5
```

15.1.15 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:

- `colspecs`: a list of pairs (tuples), giving the extents of the fixed-width fields of each line as half-open intervals [from, to]
- `widths`: a list of field widths, which can be used instead of `colspecs` if the intervals are contiguous

Consider a typical fixed-width data file:

```python
In [920]: 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:

```python
#Column specifications are a list of half-intervals
In [921]: colspecs = [(0, 6), (8, 20), (21, 33), (34, 43)]
In [922]: df = pd.read_fwf('bar.csv', colspecs=colspecs, header=None, index_col=0)
In [923]: df
Out[923]:
   1     2     3
0  id8141 360.24294 149.91019 11950.7
id1594 444.95363 166.98565 11788.4
id1849 364.13685 183.62877 11806.2
id1230 413.83612 184.3757 11916.8
id1948 502.95395 173.23716 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:

```python
#Widths are a list of integers
In [924]: widths = [6, 14, 13, 10]
```
In [925]: df = pd.read_fwf('bar.csv', widths=widths, header=None)

In [926]: df
Out[926]:
   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.

15.1.16 Files with an “implicit” index column

Consider a file with one less entry in the header than the number of data column:

In [927]: 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 [928]: pd.read_csv('foo.csv')
Out[928]:
         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 [929]: df = pd.read_csv('foo.csv', parse_dates=True)

In [930]: df.index
Out[930]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2009-01-01 00:00:00, ..., 2009-01-03 00:00:00]
Length: 3, Freq: None, Timezone: None

15.1.17 Reading DataFrame objects with MultiIndex

Suppose you have data indexed by two columns:

In [931]: print open('data/mindex_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
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",.12,.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:

```
In [932]: df = pd.read_csv("data/mindex_ex.csv", index_col=[0,1])
```

```
In [933]: df
Out[933]:
     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 [934]: df.ix[1978]
Out[934]:
     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
```

15.1.18 Automatically “sniffing” the delimiter

`read_csv` is capable of inferring delimited (not necessarily comma-separated) files. YMMV, as pandas uses the `csv.Sniffer` class of the csv module.

```
In [935]: print(open('tmp2.sv').read())
```

```
:0:1:2:3
0:0.469112299907:-0.282863344329:-1.50905850317:-1.13563273102
1:1.21211202502:-0.173214649053:0.119208711297:-1.04423596628
2:-0.861848963348:-2.10456921889:-0.494929274069:1.07180380704
3:0.721555162244:-0.70677113363:-1.03957498511:0.271859885543
4:-0.424972329789:0.567020349794:0.276232019278:-1.08740069129
5:-0.673689708088:0.113648409689:-1.47842655244:0.524987667115
6:0.40470521868:0.57704598592:-1.7150021611:-1.03926848351

15.1. CSV & Text files 291
7:-0.370646858236:-1.15789225064:-1.34431181273:0.844885141425
8:1.07576978372:-0.10904997528:1.64356307036:-1.46938795954
9:0.357020564133:-0.67460010373:-1.77690371697:-0.968913812447

In [936]: pd.read_csv('tmp2.sv')
Out[936]:
<table>
<thead>
<tr>
<th>0 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0.469112299907:-0.28286344329:-1.5090585031...</td>
</tr>
<tr>
<td>1 1.21211202502:-0.173214649053:0.11920711297...</td>
</tr>
<tr>
<td>2 -0.861848963348:-2.10456921889:-0.4949292740...</td>
</tr>
<tr>
<td>3 0.721555162244:-0.70677113363:-1.03957498511...</td>
</tr>
<tr>
<td>4 -0.424972329789:0.567020349794:0.276232019272...</td>
</tr>
<tr>
<td>5 -0.673689708088:0.113648409689:-1.4784265524...</td>
</tr>
<tr>
<td>6 0.40470521868:0.57704598592:-1.71500201611:-...</td>
</tr>
<tr>
<td>7 -0.370646858236:-1.15789225064:-1.34431181273...</td>
</tr>
<tr>
<td>8 1.07576978372:-0.10904997528:1.64356307036:-...</td>
</tr>
<tr>
<td>9 0.357020564133:-0.67460010373:-1.77690371697...</td>
</tr>
</tbody>
</table>

15.1.19 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 [937]: print open('tmp.sv').read()
|0|1|2|3|
|0|0.469112299907:-0.28286344329:-1.5090585031...|
|1|1.21211202502:-0.173214649053:0.11920711297...|
|2|-0.861848963348:-2.10456921889:-0.4949292740...|
|3|0.721555162244:-0.70677113363:-1.03957498511...|
|4|-0.424972329789:0.567020349794:0.276232019272...|
|5|-0.673689708088:0.113648409689:-1.4784265524...|
|6|0.40470521868:0.57704598592:-1.71500201611:-...|
|7|-0.370646858236:-1.15789225064:-1.34431181273...|
|8|1.07576978372:-0.10904997528:1.64356307036:-...|
|9|0.357020564133:-0.67460010373:-1.77690371697...|

In [938]: table = pd.read_table('tmp.sv', sep='|')

In [939]: table
Out[939]:
<table>
<thead>
<tr>
<th>Unnamed: 0 0 1 2 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0.469112 -0.282863 -1.509058 -1.1356327102</td>
</tr>
<tr>
<td>1 1.212112 -0.173215 0.119209 -1.04423596628</td>
</tr>
<tr>
<td>2 -0.861849 -2.104569 -0.494929 1.07180380704</td>
</tr>
<tr>
<td>3 0.721555 -0.706771 -1.039575 0.27185988543</td>
</tr>
<tr>
<td>4 -0.424972 0.567020 0.276232 1.08740069129</td>
</tr>
<tr>
<td>5 -0.673689 0.113648 -1.478426 0.524987667115</td>
</tr>
<tr>
<td>6 0.404705 0.577046 -1.715002 -1.03926848351</td>
</tr>
<tr>
<td>7 -0.370647 0.271859 0.524988 0.844885141425</td>
</tr>
<tr>
<td>8 1.075769 1.643563 -1.46938795954</td>
</tr>
<tr>
<td>9 0.357020 -0.674600 -1.776903 -0.968913812447</td>
</tr>
</tbody>
</table>

By specifying a chunksize to read_csv or read_table, the return value will be an iterable object of type TextParser:

In [940]: reader = pd.read_table('tmp.sv', sep='|', chunksize=4)
In [941]: reader
Out[941]: <pandas.io.parsers.TextFileReader at 0xcd11ad0>

In [942]: for chunk in reader:
.....: print chunk
.....:
   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
   Unnamed: 0   0   1   2   3
 0    4 -0.424972  0.567020  0.276232 -1.087401
 1    5 -0.673690  0.113648 -1.478427  0.524988
 2    6  0.404705  0.577046 -1.715002 -1.039268
 3    7 -0.370647 -1.157892 -1.344312  0.844885
   Unnamed: 0   0   1   2   3
 0    8  1.075770 -0.10905  1.643563 -1.469388
 1    9  0.357021 -0.67460  -1.776904 -0.968914

Specifying iterator=True will also return the TextParser object:

In [943]: reader = pd.read_table('tmp.sv', sep='|', iterator=True)

In [944]: reader.get_chunk(5)
Out[944]:
   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

15.1.20 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: A string path to the file to write
• nanRep: A string representation of a missing value (default '')
• 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’
• sep: Field delimiter for the output file (default ‘,’)
• encoding: a string representing the encoding to use if the contents are non-ascii, for python versions prior to 3
15.1.21 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.

15.1.22 Writing to HTML format

Dataframe object has 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.

15.2 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 described in the next section. 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:

```python
clipdf = pd.read_clipboard(delim_whitespace=True)
```

```
In [945]: clipdf
Out[945]:
   A  B  C
  x 1  4  p
  y 2  5  q
  z 3  6  r
```
15.3 Excel files

The ExcelFile class can read an Excel 2003 file using the xlrd Python module and use the same parsing code as the above to convert tabular data into a DataFrame. To use it, create the ExcelFile object:

```python
xls = ExcelFile('path_to_file.xls')
```

Then use the parse instance method with a sheetname, then use the same additional arguments as the parsers above:

```python
xls.parse('Sheet1', index_col=None, na_values=['NA'])
```

To read sheets from an Excel 2007 file, you can pass a filename with a .xlsx extension, in which case the openpyxl module will be used to read the file.

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. ExcelFile.parse takes a parse_cols keyword to allow you to specify a subset of columns to parse.

If `parse_cols` is an integer, then it is assumed to indicate the last column to be parsed.

```python
xls.parse('Sheet1', parse_cols=2, index_col=None, na_values=['NA'])
```

If `parse_cols` is a list of integers, then it is assumed to be the file column indices to be parsed.

```python
xls.parse('Sheet1', parse_cols=[0, 2, 3], index_col=None, na_values=['NA'])
```

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:

```python
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 openpyxl. The Panel class also has a `to_excel` instance method, which writes each DataFrame in the Panel to a separate sheet.

In order to write separate DataFrames to separate sheets in a single Excel file, one can use the ExcelWriter class, as in the following example:

```python
writer = ExcelWriter('path_to_file.xlsx')
df1.to_excel(writer, sheet_name='sheet1')
df2.to_excel(writer, sheet_name='sheet2')
writer.save()
```

15.4 HDF5 (PyTables)

HDFStore is a dict-like object which reads and writes pandas to the high performance HDF5 format using the excellent PyTables library.

```python
In [946]: store = HDFStore('store.h5')

In [947]: print store
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
Empty
```

Objects can be written to the file just like adding key-value pairs to a dict:
In [948]: index = date_range('1/1/2000', periods=8)
In [949]: 

In [950]: s = Series(randn(5), index=['a', 'b', 'c', 'd', 'e'])
In [951]: df = DataFrame(randn(8, 3), index=index,
                          columns=['A', 'B', 'C'])
In [952]: wp = Panel(randn(2, 5, 4), items=['Item1', 'Item2'],
                        major_axis=date_range('1/1/2000', periods=5),
                        minor_axis=['A', 'B', 'C', 'D'])

# store.put('s', s) is an equivalent method
In [953]: store['s'] = s
In [954]: store['df'] = df
In [955]: store['wp'] = wp

# the type of stored data
In [956]: 

In [957]: store.root.wp._v_attrs.pandas_type
Out[956]: 'wide'

In [958]: store
Out[958]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame (shape->[8,3])
/s series (shape->[5])
/wp wide (shape->[2,5,4])

In a current or later Python session, you can retrieve stored objects:

# store.get('df') is an equivalent method
In [959]: store['df']

Deletion of the object specified by the key

# store.remove('wp') is an equivalent method
In [960]: del store['wp']

In [961]: store
Out[961]: <class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame (shape->[8,3])
/s series (shape->[5])
Closing a Store

```python
# closing a store
In [960]: store.close()

# Working with, and automatically closing the store with the context manager.
In [961]: with get_store('store.h5') as store:
       ....:    store.keys()
       ....:
```

These 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.

### 15.4.1 Storing in 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 & query type operations are supported.

```python
In [962]: store = HDFStore('store.h5')

In [963]: df1 = df[0:4]

In [964]: df2 = df[4:]

# append data (creates a table automatically)
In [965]: store.append('df', df1)

In [966]: store.append('df', df2)

In [967]: store
Out[967]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df    frame_table  (typ->appendable,nrows->8,ncols->3,indexers->[index])

# select the entire object
In [968]: store.select('df')
Out[968]:
     A    B    C
2000-01-01 -0.362543 -0.006154 -0.923061
2000-01-02  0.895717  0.805244 -1.206412
2000-01-03  2.565646  1.431256  1.340309
2000-01-04 -1.170299 -0.226169  0.410835
2000-01-05  0.813850  0.132003 -0.827317
2000-01-06 -0.076467 -1.187678  1.130127
2000-01-07 -1.436737 -1.413681  1.607920
2000-01-08  1.024180  0.569605  0.875906

# the type of stored data
In [969]: store.root.df._v_attrs.pandas_type
Out[969]: 'frame_table'
```
15.4.2 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 without 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 [970]: store.put('foo/bar/bah', df)
In [971]: store.append('food/orange', df)
In [972]: store.append('food/apple', df)
In [973]: store
Out[973]:<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/apple frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/food/orange frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah frame (shape->[8,3])

# a list of keys are returned
In [974]: store.keys()
Out[974]: ['/df', '/food/apple', '/food/orange', '/foo/bar/bah']

# remove all nodes under this level
In [975]: store.remove('food')
In [976]: store
Out[976]:<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/foo/bar/bah frame (shape->[8,3])

15.4.3 Storing Mixed Types in a Table

Storing mixed-dtype data is supported. Strings are stored as a fixed-width using the maximum size of the appended column. Subsequent appends will truncate strings at this length. Passing `min_itemsize = { 'values' : size }` as a parameter to append will set a larger minimum for the string columns. Storing floats, strings, ints, bools, datetime64 are currently supported. For string columns, passing `nan_rep = 'nan'` to append will change the default nan representation on disk (which converts to/from `np.nan`), this defaults to `nan`.

In [977]: df_mixed = df.copy()
In [978]: df_mixed['string'] = 'string'
In [979]: df_mixed['int'] = 1
In [980]: df_mixed['bool'] = True
In [981]: df_mixed['datetime64'] = Timestamp('20010102')
In [982]: df_mixed ix[3:5, ['A','B','string','datetime64']] = np.nan
In [983]: store.append('df_mixed', df_mixed, min_itemsize = { 'values' : 50 })
**15.4.4 Storing Multi-Index DataFrames**

Storing multi-index dataframes as tables is very similar to storing/selecting from homogenous index DataFrames.

```python
In [988]: 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 [989]: df_mi = DataFrame(np.random.randn(10, 3), index=index, columns=['A', 'B', 'C'])
```

**15.4. HDF5 (PyTables)**
In [990]:

```
Out[990]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo one</td>
<td>0.896171</td>
<td>-0.487602</td>
</tr>
<tr>
<td>two</td>
<td>-2.182937</td>
<td>0.380396</td>
</tr>
<tr>
<td>three</td>
<td>0.432390</td>
<td>1.519970</td>
</tr>
<tr>
<td>bar one</td>
<td>0.600178</td>
<td>0.274230</td>
</tr>
<tr>
<td>two</td>
<td>-0.023688</td>
<td>2.410179</td>
</tr>
<tr>
<td>baz two</td>
<td>0.206053</td>
<td>-0.251905</td>
</tr>
<tr>
<td>three</td>
<td>1.063327</td>
<td>1.266143</td>
</tr>
<tr>
<td>qux one</td>
<td>-0.863838</td>
<td>0.408204</td>
</tr>
<tr>
<td>two</td>
<td>-0.025747</td>
<td>-0.988387</td>
</tr>
<tr>
<td>three</td>
<td>1.262731</td>
<td>1.289997</td>
</tr>
</tbody>
</table>
```

In [991]: store.append('df_mi', df_mi)

In [992]: store.select('df_mi')

Out[992]:

```
Out[992]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo one</td>
<td>0.896171</td>
<td>-0.487602</td>
</tr>
<tr>
<td>two</td>
<td>-2.182937</td>
<td>0.380396</td>
</tr>
<tr>
<td>three</td>
<td>0.432390</td>
<td>1.519970</td>
</tr>
<tr>
<td>bar one</td>
<td>0.600178</td>
<td>0.274230</td>
</tr>
<tr>
<td>two</td>
<td>-0.023688</td>
<td>2.410179</td>
</tr>
<tr>
<td>baz two</td>
<td>0.206053</td>
<td>-0.251905</td>
</tr>
<tr>
<td>three</td>
<td>1.063327</td>
<td>1.266143</td>
</tr>
<tr>
<td>qux one</td>
<td>-0.863838</td>
<td>0.408204</td>
</tr>
<tr>
<td>two</td>
<td>-0.025747</td>
<td>-0.988387</td>
</tr>
<tr>
<td>three</td>
<td>1.262731</td>
<td>1.289997</td>
</tr>
</tbody>
</table>
```

# the levels are automatically included as data columns

In [993]: store.select('df_mi', Term('foo=bar'))

Out[993]:

```
Out[993]:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>foo bar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bar one</td>
<td>0.600178</td>
<td>0.274230</td>
</tr>
<tr>
<td>two</td>
<td>-0.023688</td>
<td>2.410179</td>
</tr>
</tbody>
</table>
```

### 15.4.5 Querying a Table

`select` and `delete` operations have an optional criteria 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.

- ‘index’ and ‘columns’ are supported indexers of a DataFrame
- ‘major_axis’, ‘minor_axis’, and ‘items’ are supported indexers of the Panel

Valid terms can be created from `dict`, `list`, `tuple`, or `string`. Objects can be embedded as values. Allowed operations are: `<`, `<=`, `>`, `>=`, `=`. `=` will be inferred as an implicit set operation (e.g. if 2 or more values are provided). The following are all valid terms.

- `dict(field = 'index', op = '>', value = '20121114')`
- `('index', '>', '20121114')`
- `'index > 20121114'`
• ('index', '>', datetime(2012,11,14))
• ('index', ['20121114', '20121115'])
• ('major_axis', '=', Timestamp('2012/11/14'))
• ('minor_axis', ['A', 'B'])

Queries are built up using a list of Terms (currently only **anding** of terms is supported). An example query for a panel might be specified as follows: ['major_axis>20000102', ('minor_axis', '=', ['A', 'B'])]. This is roughly translated to: **major_axis must be greater than the date 20000102 and the minor_axis must be A or B**

**In [994]:** store.append('wp', wp)

**In [995]:** store
**Out[995]:**
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df_mi frame_table (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->[bar,foo])
/df_mixed frame_table (typ->appendable,nrows->8,ncols->7,indexers->[index])
/wp wide_table (typ->appendable,nrows->20,ncols->2,indexers->[major_axis,minor_axis])
/foo/bar/bah frame (shape->[8,3])

**In [996]:** store.select('wp', [ Term('major_axis>20000102'), Term('minor_axis', '=', ['A', 'B']) ])
**Out[996]:**
<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 to filter a list of the return columns, this is equivalent to passing a Term('columns',list_of_columns_to_filter)

**In [997]:** store.select('df', columns = ['A', 'B'])
**Out[997]:**
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000-01-01</td>
<td>-0.362543</td>
<td>-0.006154</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>0.895717</td>
<td>0.805244</td>
</tr>
<tr>
<td>2000-01-03</td>
<td>2.565646</td>
<td>1.431256</td>
</tr>
<tr>
<td>2000-01-04</td>
<td>-1.170299</td>
<td>-0.226169</td>
</tr>
<tr>
<td>2000-01-05</td>
<td>0.813850</td>
<td>0.132003</td>
</tr>
<tr>
<td>2000-01-06</td>
<td>-0.076467</td>
<td>-1.187678</td>
</tr>
<tr>
<td>2000-01-07</td>
<td>1.436737</td>
<td>-1.413681</td>
</tr>
<tr>
<td>2000-01-08</td>
<td>1.024180</td>
<td>0.569605</td>
</tr>
</tbody>
</table>

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.

```python
# this is effectively what the storage of a Panel looks like
In [998]: wp.to_frame()
```
**Out[998]:**

<table>
<thead>
<tr>
<th></th>
<th>Item1</th>
<th>Item2</th>
</tr>
</thead>
<tbody>
<tr>
<td>major</td>
<td>minor</td>
<td></td>
</tr>
<tr>
<td>2000-01-01</td>
<td>A</td>
<td>-2.211372</td>
</tr>
<tr>
<td>B</td>
<td>0.974466</td>
<td>0.176444</td>
</tr>
<tr>
<td>C</td>
<td>-2.006747</td>
<td>0.403310</td>
</tr>
<tr>
<td>D</td>
<td>-0.410001</td>
<td>-0.154951</td>
</tr>
<tr>
<td>2000-01-02</td>
<td>A</td>
<td>-0.078638</td>
</tr>
</tbody>
</table>

15.4. HDF5 (PyTables) 301
# limiting the search

In [999]: store.select('wp', [Term('major_axis>20000102'), Term('minor_axis', '=', ['A', 'B'])], start=0, stop=10)

Out[999]:
<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

15.4.6 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. Indexes are automagically created (starting 0.10.1) on the indexables and any data columns you specify. This behavior can be turned off by passing index=False to append.

# we have automagically already created an index (in the first section)
In [1000]: i = store.root.df.table.cols.index.index

In [1001]: i.optlevel, i.kind
Out[1001]: (6, 'medium')

# change an index by passing new parameters
In [1002]: store.create_table_index('df', optlevel = 9, kind = 'full')

In [1003]: i = store.root.df.table.cols.index.index

In [1004]: i.optlevel, i.kind
Out[1004]: (9, 'full')

15.4.7 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 [1005]: df_dc = df.copy()

In [1006]: df_dc['string'] = 'foo'

In [1007]: df_dc.ix[4:6,'string'] = np.nan

In [1008]: df_dc.ix[7:9,'string'] = 'bar'

In [1009]: df_dc['string2'] = 'cool'

In [1010]: df_dc
Out[1010]:
   A       B       C     string  string2
0  2000-01-01 -0.362543 -0.006154    foo     cool
1  2000-01-02  0.895717  0.805244    foo     cool
2  2000-01-03  2.565646  1.431256    foo     cool
3  2000-01-04 -1.170299 -0.226169    foo     cool
4  2000-01-05  0.813850  0.132003  -0.827317    NaN     cool
5  2000-01-06 -0.076467 -1.187678  1.130127    NaN     cool
6  2000-01-07 -1.436737 -1.413681  1.607920    foo     cool
7  2000-01-08  1.024180  0.569605  0.875906    bar     cool

# on-disk operations
In [1011]: store.append('df_dc', df_dc, data_columns = ['B','C','string','string2'])

In [1012]: store.select('df_dc', [Term('B>0')])
Out[1012]:
   A       B       C     string  string2
0  2000-01-02  0.895717  0.805244    foo     cool
1  2000-01-03  2.565646  1.431256    foo     cool
2  2000-01-05  0.813850  0.132003  -0.827317    NaN     cool
3  2000-01-08  1.024180  0.569605  0.875906    bar     cool

# getting creative
In [1013]: store.select('df_dc', [ 'B > 0', 'C > 0', 'string == foo' ])
Out[1013]:
   A       B       C     string  string2
0  2000-01-03  2.565646  1.431256    foo     cool

# this is in-memory version of this type of selection
In [1014]: df_dc[(df_dc.B > 0) & (df_dc.C > 0) & (df_dc.string == 'foo')]
Out[1014]:
   A       B       C     string  string2
0  2000-01-03  2.565646  1.431256    foo     cool

# we have automagically created this index and that the B/C/string/string2 columns are stored separately
In [1015]: store.root.df_dc.table
Out[1015]:
'/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),
   "C": Float64Col(shape=(), dflt=0.0, pos=3),
   "string": StringCol(itemsize=3, shape=(), dflt='', pos=4),
   "string2": StringCol(itemsize=4, shape=(), dflt='', pos=5)}
byteorder := 'little'
chunkshape := (1680,)
There is some performance degradation by making lots of columns into \textit{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!)

### 15.4.8 Advanced Queries

**Unique**

To retrieve the \textit{unique} values of an indexable or data column, use the method \texttt{unique}. This will, for example, enable you to get the index very quickly. Note \texttt{nan} are excluded from the result set.

In [1016]: store.unique('df_dc','index')
Out[1016]:
<class 'pandas.tseries.index.DatetimeIndex'>
[2000-01-01 00:00:00, ..., 2000-01-08 00:00:00]
Length: 8, Freq: None, Timezone: None

In [1017]: store.unique('df_dc','string')
Out[1017]: Index([bar, foo], dtype=object)

**Replicating or**

\texttt{not} and \texttt{and} or \texttt{or} conditions are unsupported at this time; however, \texttt{or} operations are easy to replicate, by repeatedly applying the criteria to the table, and then \texttt{concat} the results.

In [1018]: crit1 = [ Term('B>0'), Term('C>0'), Term('string=foo') ]

In [1019]: crit2 = [ Term('B<0'), Term('C>0'), Term('string=foo') ]

In [1020]: concat([ store.select('df_dc',c) for c in [crit1, crit2] ])
Out[1020]:
   A  B   C string string2
0 2.565646 1.431256 1.340309  foo  cool
1 -1.170299 -0.226169 0.410835  foo  cool
2 -1.436737 -1.413681 1.607920  foo  cool

**Storer Object**

If you want to inspect the stored object, retrieve via \texttt{get_storer}. You could use this programmatically to say get the number of rows in an object.

In [1021]: store.get_storer('df_dc').nrows
Out[1021]: 8

### 15.4.9 Multiple Table Queries

New in 0.10.1 are the methods \texttt{append_to_multiple} and \texttt{select_as_multiple}, that 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 that are indexed the same the
selector table. You can then perform a very fast query on the selector table, yet get lots of data back. This method works similar to having a very wide-table, but is more efficient in terms of queries.

Note, **THE USER IS RESPONSIBLE FOR SYNCHRONIZING THE TABLES.** This means, append to the tables in the same order; append_to_multiple splits a single object to multiple tables, given a specification (as a dictionary). This dictionary is a mapping of the table names to the 'columns' you want included in that table. Pass a None for a single table (optional) to let it have the remaining columns. The argument selector defines which table is the selector table.

```
In [1022]: df_mt = DataFrame(randn(8, 6), index=date_range('1/1/2000', periods=8),
                    columns=['A', 'B', 'C', 'D', 'E', 'F'])
```

```
In [1023]: df_mt['foo'] = 'bar'
```

# you can also create the tables individually
```
In [1024]: store.append_to_multiple({'df1_mt': ['A','B'], 'df2_mt': None}, df_mt, selector = 'df1_mt')
```

```
In [1025]: store
Out[1025]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

```
/df frame_table (typ->appendable,nrows->8,ncols->3,indexers->[index])
/df1_mt frame_table (typ->appendable,nrows->8,ncols->2,indexers->[index],dc->[A,B])
/df2_mt frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index])
/df_dc frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[B,C,string,string2])
/df_mi frame_table (typ->appendable_multi,nrows->10,ncols->5,indexers->[index],dc->)
/df_mixed frame_table (typ->appendable,nrows->8,ncols->7,indexers->[index])
/wp wide_table (typ->appendable,nrows->20,ncols->2,indexers->[major_axis,minor_axis])
/foo/bar/bah frame (shape->[8,3])
```

# indiviual tables were created
```
In [1026]: store.select('df1_mt')
Out[1026]:
```
   A   B
2000-01-01 -0.055758  0.536580
2000-01-02 -0.281461  0.030711
2000-01-03 -0.064034 -1.282782
2000-01-04  0.583787  0.221471
2000-01-05 -0.845696 -1.340896
2000-01-06  0.888782  0.228440
2000-01-07 -1.066969 -0.303421
2000-01-08  1.574159  1.588931
```

```
In [1027]: store.select('df2_mt')
Out[1027]:
```
```
   C   D  E   F  foo
2000-01-01 -0.489682  0.369374 -0.034571 -2.484478 {'bar'}
2000-01-02  0.109121  1.126203 -0.977349  1.474071 {'bar'}
2000-01-03  0.781836 -1.071357  0.441153  2.353925 {'bar'}
2000-01-04 -0.744471  0.758527  1.729689 -0.964980 {'bar'}
2000-01-05  1.846883 -1.328865  1.682706 -1.717693 {'bar'}
2000-01-06  0.901805  1.171216  0.520260 -1.197071 {'bar'}
2000-01-07 -0.858447  0.306996 -0.028665  0.384316 {'bar'}
2000-01-08  0.476720  0.473424 -0.242861 -0.014805 {'bar'}
```

# as a multiple
```
In [1028]: store.select_as_multiple(['df1_mt','df2_mt'], where = ['A>0','B>0'], selector = 'df1_mt')
Out[1028]:
```

15.4. HDF5 (PyTables)
15.4.10 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 deletion speed, it pays 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.

```python
# returns the number of rows deleted
In [1029]: store.remove('wp', 'major_axis>20000102' )
Out[1029]: 12

In [1030]: store.select('wp')
Out[1030]:
<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
```

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 clean the file, use ptrepack (see below).

15.4.11 Compression

PyTables allows the stored data to be compressed. This applies to all kinds of stores, not just tables.

- Pass `complevel=int` for a compression level (1-9, with 0 being no compression, and the default)
• Pass `complib=lib` where `lib` is any of `zlib`, `bzip2`, `lzo`, `blosc` for whichever compression library you prefer.

HDFStore will use the file based compression scheme if no overriding `complib` or `complevel` options are provided. `blosc` offers very fast compression, and is my most used. Note that `lzo` and `bzip2` may not be installed (by Python) by default.

Compression for all objects within the file

• `store_compressed = HDFStore('store_compressed.h5', complevel=9, complib='blosc')`

Or on-the-fly compression (this only applies to tables). You can turn off file compression for a specific table by passing `complevel=0`

• `store.append('df', df, complib='zlib', complevel=5)`

**ptrepack**

PyTables offer better write performance when compressed after writing them, 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.

### 15.4.12 Notes & Caveats

• Once a table is created its items (Panel) / columns (DataFrame) are fixed; only exactly the same columns can be appended

• You can not append/select/delete to a non-table (table creation is determined on the first append, or by passing `table=True` in a put operation)

• 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 issue <https://github.com/pydata/pandas/issues/2397> for more information.

• PyTables only supports fixed-width string columns in tables. The sizes of a string based indexing column (e.g. `columns` or `minor_axis`) are determined as the maximum size of the elements in that axis or by passing the parameter `min_itemsize` on the first table creation (`min_itemsize` can be an integer or a dict of column name to an integer). If subsequent appends introduce elements in the indexing axis that are larger than the supported indexer, an Exception will be raised (otherwise you could have a silent truncation of these indexers, leading to loss of information). Just to be clear, this fixed-width restriction applies to **indexables** (the indexing columns) and **string values** in a mixed_type table.

```python
In [1031]: store.append('wp_big_strings', wp, min_itemsize = { 'minor_axis' : 30 })
In [1032]: wp = wp.rename_axis(lambda x: x + '_big_strings', axis=2)
In [1033]: store.append('wp_big_strings', wp)
In [1034]: store.select('wp_big_strings')
Out[1034]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 8 (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_big_strings

# we have provided a minimum minor_axis indexable size
In [1035]: store.root.wp_big_strings.table
Out[1035]:
/wp_big_strings/table (Table(40,)) ''
description := {
  "major_axis": Int64Col(shape=(), dflt=0, pos=0),
  "minor_axis": StringCol(itemsize=30, shape=(), dflt='', pos=1),
  "values_block_0": Float64Col(shape=(2,), dflt=0.0, pos=2)}
byteorder := 'little'
chunkshape := (1213,)
autoIndex := True
colindexes := {
  "major_axis": Index(6, medium, shuffle, zlib(1)).is_CSI=False,
  "minor_axis": Index(6, medium, shuffle, zlib(1)).is_CSI=False

15.4.13 External Compatibility

HDFStore write storer objects in specific formats suitable for producing loss-less roundtrips 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. Create a table format store like this:

In [1036]: store_export = HDFStore('export.h5')
In [1037]: store_export.append('df_dc',df_dc,data_columns=df_dc.columns)
In [1038]: store_export
Out[1038]:
<class 'pandas.io.pytables.HDFStore'>
File path: export.h5
/df_dc frame_table (typ->appendable,nrows->8,ncols->5,indexers->[index],dc->[A,B,C,string,string2])

15.4.14 Backwards Compatibility

0.10.1 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. HDFStore will issue a warning if you try to use a prior-version format file. You must read in the entire file and write it out using the new format, using the method copy to take advantage of the updates. The group attribute pandas_version contains the version information. copy takes a number of options, please see the docstring.

# a legacy store
In [1039]: legacy_store = HDFStore(legacy_file_path,'r')
In [1040]: legacy_store
Out[1040]:
<class 'pandas.io.pytables.HDFStore'>
File path: /home/wesm/code/pandas/doc/source/_static/legacy_0.10.h5
/a series (shape->[30])
/b frame (shape->[30,4])
/df1_mixed frame_table [0.10.0] (typ->appendable,nrows->30,ncols->11,indexers->[index],dc->[A,B,C,string,string2])
/pl1_mixed wide_table [0.10.0] (typ->appendable,nrows->120,ncols->9,indexers->[major_axis,minor_axis])
/p4d_mixed ndim_table [0.10.0] (typ->appendable,nrows->360,ncols->9,indexers->[items,major_axis,minor_axis])
# copy (and return the new handle)
In [1041]: new_store = legacy_store.copy('store_new.h5')

In [1042]: new_store
Out[1042]:
<class 'pandas.io.pytables.HDFStore'>
File path: store_new.h5
/a    series (shape->[30])
/b    frame (shape->[30,4])
/df1_mixed frame_table (typ->appendable,nrows->30,ncols->11,indexers->[index])
/pl_mixed wide_table (typ->appendable,nrows->120,ncols->9,indexers->[major_axis,minor_axis])
/p4d_mixed wide_table (typ->appendable,nrows->360,ncols->9,indexers->[items,major_axis,minor_axis])
/... bar wide (shape->[3,30,4])

In [1043]: new_store.close()

15.4.15 Performance

- Tables come with a writing performance penalty as compared to regular 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=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 expectedrows that PyTables will expected. 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 <http://stackoverflow.com/questions/14355151/how-to-make-pandas-hdfstore-put-operation-faster/14370190#14370190> for more information and some solutions.

15.4.16 Experimental

HDFStore supports Panel4D storage.

In [1044]: p4d = Panel4D({ 'l1' : wp })

In [1045]: p4d
Out[1045]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 1 (labels) x 2 (items) x 5 (major_axis) x 4 (minor_axis)
Labels axis: l1 to l1
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_big_strings to D_big_strings

In [1046]: store.append('p4d', p4d)

In [1047]: store
Out[1047]:

15.4. HDF5 (PyTables)
These, by default, index the three axes `items`, `major_axis`, `minor_axis`. On an `AppendableTable` it is possible to setup with the first append a different indexing scheme, depending on how you want to store your data. Pass the `axes` keyword with a list of dimension (currently must by exactly 1 less than the total dimensions of the object). This cannot be changed after table creation.

```
In [1048]: store.append('p4d2', p4d2, axes = ['labels','major_axis','minor_axis'])
```

```
In [1049]: store
Out[1049]:
<class 'pandas.io.pytables.HDFStore'>
File path: store.h5
```

```
In [1050]: store.select('p4d2', [ Term('labels=11'), Term('items=Item1'), Term('minor_axis=A_big_strings') ])
```

```
Out[1050]:
<class 'pandas.core.panelnd.Panel4D'>
Dimensions: 1 (labels) x 1 (items) x 5 (major_axis) x 1 (minor_axis)
Labels axis: 11 to 11
Items axis: Item1 to Item1
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
Minor_axis axis: A_big_strings to A_big_strings
```

### 15.5 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. There wrappers only support the Python database adapters which respect the Python DB-API.

Suppose you want to query some data with different types from a table such as:
Functions from pandas.io.sql can extract some data into a DataFrame. In the following example, we use SQLite SQL database engine. You can use a temporary SQLite database where data are stored in “memory”. Just do:

```python
import sqlite3
from pandas.io import sql
# Create your connection.
cnx = sqlite3.connect(':memory:)
```

Let data be the name of your SQL table. With a query and your database connection, just use the `read_frame()` function to get the query results into a DataFrame:

```python
In [1051]: sql.read_frame("SELECT * FROM data;", cnx)
Out[1051]:
id  date  Col_1  Col_2  Col_3
0  26  2010-10-18 00:00:00 X  27.50  1
1  42  2010-10-19 00:00:00 Y -12.50  0
2  63  2010-10-20 00:00:00 Z  5.73  1
```

You can also specify the name of the column as the DataFrame index:

```python
In [1052]: sql.read_frame("SELECT * FROM data;", cnx, index_col='id')
Out[1052]:
date  Col_1  Col_2  Col_3
id 2010-10-18 00:00:00 26 X 27.50 1
42 2010-10-19 00:00:00 Y -12.50 0
63 2010-10-20 00:00:00 Z 5.73 1
```

Of course, you can specify more “complex” query.

```python
In [1053]: sql.read_frame("SELECT id, Col_1, Col_2 FROM data WHERE id = 42;", cnx)
Out[1053]:
id Col_1  Col_2
0 42  Y -12.5
```

There are a few other available functions:

- `tquery` returns list of tuples corresponding to each row.
- `uquery` does the same thing as `tquery`, but instead of returning results, it returns the number of related rows.
- `write_frame` writes records stored in a DataFrame into the SQL table.
- `has_table` checks if a given SQLite table exists.

Note: For now, writing your DataFrame into a database works only with SQLite. Moreover, the index will currently be dropped.
SPARSE DATA STRUCTURES

We have implemented “sparse” versions of Series, DataFrame, and Panel. These are not sparse in the typical “mostly 0”. You can view these objects as being “compressed” where any data matching a specific value (NaN/missing by default, though any value can be chosen) is omitted. A special SparseIndex object tracks where data has been “sparsified”. This will make much more sense in an example. All of the standard pandas data structures have a `to_sparse` method:

```python
In [1276]: ts = Series(randn(10))
In [1277]: ts[2:-2] = np.nan
In [1278]: sts = ts.to_sparse()
```

```python
In [1279]: sts
Out[1279]:
0   0.469112
1  -0.282863
2      NaN
3      NaN
4      NaN
5      NaN
6      NaN
7      NaN
8  -0.861849
9  -2.104569
```

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`:

```python
In [1280]: ts.fillna(0).to_sparse(fill_value=0)
```

```python
Out[1280]:
0   0.469112
1  -0.282863
2     0.000000
3     0.000000
4     0.000000
5     0.000000
6     0.000000
7     0.000000
8  -0.861849
9  -2.104569
```
The sparse objects exist for memory efficiency reasons. Suppose you had a large, mostly NA DataFrame:

```
In [1281]: df = DataFrame(randn(10000, 4))
In [1282]: df.ix[:9998] = np.nan
In [1283]: sdf = df.to_sparse()
In [1284]: sdf
```

```
<class 'pandas.sparse.frame.SparseDataFrame'>
Int64Index: 10000 entries, 0 to 9999
Data columns:
0 1 non-null values
1 1 non-null values
2 1 non-null values
3 1 non-null values
dtypes: float64(4)
```

```
In [1285]: sdf.density
```

```
Out[1285]: 0.0001
```

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 [1286]: sts.to_dense()
```

```
Out[1286]:
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
```

### 16.1 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 [1287]: arr = np.random.randn(10)
In [1288]: arr[2:5] = np.nan; arr[7:8] = np.nan
In [1289]: sparr = SparseArray(arr)
```
In [1290]: sparr
Out[1290]:
SparseArray([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453,
            nan, 0.606 , 1.3342])
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)

Like the indexed objects (SparseSeries, SparseDataFrame, SparsePanel), a SparseArray can be converted back to
a regular ndarray by calling to_dense:

In [1291]: sparr.to_dense()
Out[1291]:
array([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453,
            nan, 0.606 , 1.3342])

16.2 SparseList

SparseList is a list-like data structure for managing a dynamic collection of SparseArrays. To create one, simply
call the SparseList constructor with a fill_value (defaulting to NaN):

In [1292]: spl = SparseList()

In [1293]: spl
Out[1293]:
<pandas.sparse.list.SparseList object at 0xea14590>

The two important methods are append and to_array. append can accept scalar values or any 1-dimensional
sequence:

In [1294]: spl.append(np.array([1., nan, nan, 2., 3.]))

In [1295]: spl.append(5)

In [1296]: spl.append(sparr)

In [1297]: spl
Out[1297]:
<pandas.sparse.list.SparseList object at 0xea14590>
SparseArray([ 1., nan, nan, 2., 3.])
IntIndex
Indices: array([0, 3, 4], dtype=int32)
SparseArray([ 5.])
IntIndex
Indices: array([0], dtype=int32)
SparseArray([-1.9557, -1.6589, nan, nan, nan, 1.1589, 0.1453,
            nan, 0.606 , 1.3342])
IntIndex
Indices: array([0, 1, 5, 6, 8, 9], dtype=int32)

As you can see, all of the contents are stored internally as a list of memory-efficient SparseArray objects. Once
you’ve accumulated all of the data, you can call to_array to get a single SparseArray with all the data:

In [1298]: spl.to_array()
Out[1298]:
SparseArray([ 1. , nan, nan, 2. , 3. , 5. , -1.9557,
           -1.6589, nan, nan, nan, 1.1589, 0.1453, nan,
16.3 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.
CAVEATS AND GOTCHAS

17.1 NaN, Integer NA values and NA type promotions

17.1.1 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
- 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 isnull and notnull 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.

17.1.2 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 [488]: s = Series([1, 2, 3, 4, 5], index=list('abcde'))

In [489]: s
Out[489]:
   a  1
   b  2
   c  3
   d  4
   e  5
   dtype: int64

In [490]: s.dtype
Out[490]: dtype('int64')

In [491]: s2 = s.reindex(['a', 'b', 'c', 'f', 'u'])

In [492]: s2
Out[492]:
   a  1
b  2
c  3
f  NaN
u  NaN
dtype: float64

In [493]: s2.dtype
Out[493]: 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.

### 17.1.3 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. These are summarized by this table:

<table>
<thead>
<tr>
<th>Typeclass</th>
<th>Promotion dtype for storing NAs</th>
</tr>
</thead>
<tbody>
<tr>
<td>floating</td>
<td>no change</td>
</tr>
<tr>
<td>object</td>
<td>no change</td>
</tr>
<tr>
<td>integer</td>
<td>cast to float64</td>
</tr>
<tr>
<td>boolean</td>
<td>cast to object</td>
</tr>
</tbody>
</table>

While this may seem like a heavy trade-off, in practice I have found very few cases where this is an issue in practice. Some explanation for the motivation here in the next section.

### 17.1.4 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:

<table>
<thead>
<tr>
<th>Typeclass</th>
<th>Dtypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>numpy.floating</td>
<td>float16, float32, float64, float128</td>
</tr>
<tr>
<td>numpy.integer</td>
<td>int8, int16, int32, int64</td>
</tr>
<tr>
<td>numpy.unsignedinteger</td>
<td>uint8, uint16, uint32, uint64</td>
</tr>
<tr>
<td>numpy.object_</td>
<td>object_</td>
</tr>
<tr>
<td>numpy.bool_</td>
<td>bool_</td>
</tr>
<tr>
<td>numpy.character</td>
<td>string_, unicode_</td>
</tr>
</tbody>
</table>

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.
17.2 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 .ix. The following code will generate exceptions:

```python
s = Series(range(5))
s[-1]
df = DataFrame(np.random.randn(5, 4))
df
df.ix[-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).

17.3 Label-based slicing conventions

17.3.1 Non-monotonic indexes require exact matches

17.3.2 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:

```python
In [494]: s = Series(randn(6), index=list('abcdef'))
Out[494]:
   a    1.337122
   b   -1.531095
   c    1.331458
   d   -0.571329
   e   -0.026671
   f   -1.085663
   dtype: float64
```

Suppose we wished to slice from `c` to `e`, using integers this would be

```python
In [496]: s[2:5]
Out[496]:
   c    1.331458
   d   -0.571329
   e   -0.026671
   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:

```python
s.ix['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 decision to make label-based slicing include both endpoints:
In [497]:  s.ix['c':'e']
Out[497]:
c  1.331458
  d -0.571329
e -0.026671
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.

17.4 Miscellaneous indexing gotchas

17.4.1 Reindex versus ix gotchas

Many users will find themselves using the `ix` indexing capabilities as a concise means of selecting data from a pandas object:

```python
In [498]:  df = DataFrame(randn(6, 4), columns=['one', 'two', 'three', 'four'],
.....:                  index=list('abcdef'))
.....:
```

```python
In [499]:  df
Out[499]:
one two three four
a -1.114738 -0.058216 -0.486768 1.685148
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
d  0.184735 -0.255069 -0.271020  1.288393
e  0.294633 -1.165787  0.846974 -0.685597
f  0.609099 -0.303961  0.625555 -0.059268
```

```python
In [500]:  df.ix[['b', 'c', 'e']]
Out[500]:
one two three four
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
e  0.294633 -1.165787  0.846974 -0.685597
```

This is, of course, completely equivalent in this case to using the `reindex` method:

```python
In [501]:  df.reindex(['b', 'c', 'e'])
Out[501]:
one two three four
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
e  0.294633 -1.165787  0.846974 -0.685597
```

Some might conclude that `ix` and `reindex` are 100% equivalent based on this. This is indeed true except in the case of integer indexing. For example, the above operation could alternately have been expressed as:

```python
In [502]:  df.ix[[1, 2, 4]]
Out[502]:
one two three four
b  0.112572 -1.495309  0.898435 -0.148217
c -1.596070  0.159653  0.262136  0.036220
e  0.294633 -1.165787  0.846974 -0.685597
If you pass \([1, 2, 4]\) to `reindex` you will get another thing entirely:

```python
In [503]: df.reindex([1, 2, 4])
Out[503]:
   one  two  three  four
0  NaN  NaN  NaN  NaN
1  NaN  NaN  NaN  NaN
2  NaN  NaN  NaN  NaN
3  NaN  NaN  NaN  NaN
```

So it’s important to remember that `reindex` is **strict label indexing only**. This can lead to some potentially surprising results in pathological cases where an index contains, say, both integers and strings:

```python
In [504]: s = Series([1, 2, 3], index=['a', 0, 1])
In [505]: s
Out[505]:
a 1
0 2
1 3
dtype: int64
In [506]: s.ix[[0, 1]]
Out[506]:
0 2
1 3
dtype: int64
In [507]: s.reindex([0, 1])
Out[507]:
0 2
1 3
dtype: int64
```

Because the index in this case does not contain solely integers, `ix` falls back on integer indexing. By contrast, `reindex` only looks for the values passed in the index, thus finding the integers \(0\) and \(1\). While it would be possible to insert some logic to check whether a passed sequence is all contained in the index, that logic would exact a very high cost in large data sets.

### 17.4.2 Reindex potentially changes underlying Series dtype

The use of `reindex_like` can potentially change the `dtype` of a `Series`.

```python
series = pandas.Series([1, 2, 3])
x = pandas.Series([True])
x.dtype
x = pandas.Series([True]).reindex_like(series)
x.dtype
```

This is because `reindex_like` 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.
17.5 Timestamp limitations

17.5.1 Minimum and maximum timestamps

Since pandas represents timestamps in nanosecond resolution, the timespan that can be represented using a 64-bit integer is limited to approximately 584 years:

```
In [508]: begin = Timestamp(-9223372036854775809L)
```

```
In [509]: begin
Out[509]: <Timestamp: 1677-09-22 00:12:43.145224191>
```

```
In [510]: end = Timestamp(np.iinfo(np.int64).max)
```

```
In [511]: end
```

If you need to represent time series data outside the nanosecond timespan, use PeriodIndex:

```
In [512]: span = period_range('1215-01-01', '1381-01-01', freq='D')
```

```
In [513]: span
Out[513]:<class 'pandas.tseries.period.PeriodIndex'>
freq: D
[1215-01-01, ..., 1381-01-01]
length: 60632
```

17.6 Parsing Dates from Text Files

When parsing multiple text file columns into a single date column, the new date column is prepended to the data and then `index_col` specification is indexed off of the new set of columns rather than the original ones:

```
In [514]: 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 [515]: date_spec = {'nominal': [1, 2], 'actual': [1, 3]}
```

```
In [516]: df = read_csv('tmp.csv', header=None,  
                      parse_dates=date_spec,  
                      keep_date_col=True,  
                      index_col=0)
```

```
# index_col=0 refers to the combined column "nominal" and not the original  
# first column of ‘KORD’ strings
```

```
In [517]: df
Out[517]:
          actual 0 1 2 3 4
nominal
17.7 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.
CHAPTER
EIGHTEEN

RPY2 / R INTERFACE

Note: This is all highly experimental. I would like to get more people involved with building a nice RPy2 interface for pandas.

If your computer has R and rpy2 (> 2.2) installed (which will be left to the reader), you will be able to leverage the below functionality. On Windows, doing this is quite an ordeal at the moment, but users on Unix-like systems should find it quite easy. rpy2 evolves in time, and is currently reaching its release 2.3, while the current interface is designed for the 2.2.x series. We recommend to use 2.2.x over other series unless you are prepared to fix parts of the code, yet the rpy2-2.3.0 introduces improvements such as a better R-Python bridge memory management layer so I might be a good idea to bite the bullet and submit patches for the few minor differences that need to be fixed.

# if installing for the first time
hg clone http://bitbucket.org/lgautier/rpy2
cd rpy2
hg pull
hg update version_2.2.x
sudo python setup.py install

Note: To use R packages with this interface, you will need to install them inside R yourself. At the moment it cannot install them for you.

Once you have done installed R and rpy2, you should be able to import pandas.rpy.common without a hitch.

18.1 Transferring R data sets into Python

The load_data function retrieves an R data set and converts it to the appropriate pandas object (most likely a DataFrame):

In [1214]: import pandas.rpy.common as com

In [1215]: infert = com.load_data('infert')

In [1216]: infert.head()
Out[1216]:
<table>
<thead>
<tr>
<th>education</th>
<th>age</th>
<th>parity</th>
<th>induced</th>
<th>case</th>
<th>spontaneous</th>
<th>stratum</th>
<th>pooled.stratum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-5yrs</td>
<td>26</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0-5yrs</td>
<td>42</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>0-5yrs</td>
<td>39</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
18.2 Converting DataFrames into R objects

New in version 0.8. Starting from pandas 0.8, there is experimental support to convert DataFrames into the equivalent R object (that is, `data.frame`):

```
In [1217]: from pandas import DataFrame

In [1218]: df = DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]},
               index=['one', 'two', 'three'])

In [1219]: r_dataframe = com.convert_to_r_dataframe(df)

In [1220]: print type(r_dataframe)
<class 'rpy2.robjects.vectors.DataFrame'>

In [1221]: print r_dataframe
   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.

You can also use `convert_to_r_matrix` to obtain a Matrix instance, but bear in mind that it will only work with homogeneously-typed DataFrames (as R matrices bear no information on the data type):

```
In [1222]: r_matrix = com.convert_to_r_matrix(df)

In [1223]: print type(r_matrix)
<class 'rpy2.robjects.vectors.Matrix'>

In [1224]: print r_matrix
   A  B  C
one 1  4  7
two 2  5  8
three 3  6  9
```

18.3 Calling R functions with pandas objects

18.4 High-level interface to R estimators
CHAPTER
NINETEEN

RELATED PYTHON LIBRARIES

19.1 la (larry)

Keith Goodman’s excellent labeled array package is very similar to pandas in many regards, though with some key differences. The main philosophical design difference is to be a wrapper around a single NumPy ndarray object while adding axis labeling and label-based operations and indexing. Because of this, creating a size-mutable object with heterogeneous columns (e.g. DataFrame) is not possible with the la package.

• Provide a single n-dimensional object with labeled axes with functionally analogous data alignment semantics to pandas objects
• Advanced / label-based indexing similar to that provided in pandas but setting is not supported
• Stays much closer to NumPy arrays than pandas– larry objects must be homogeneously typed
• GroupBy support is relatively limited, but a few functions are available: group_mean, group_median, and group_ranking
• It has a collection of analytical functions suited to quantitative portfolio construction for financial applications
• It has a collection of moving window statistics implemented in Bottleneck

19.2 statsmodels

The main statistics and econometrics library for Python. pandas has become a dependency of this library.

19.3 scikits.timeseries

scikits.timeseries provides a data structure for fixed frequency time series data based on the numpy.MaskedArray class. For time series data, it provides some of the same functionality to the pandas Series class. It has many more functions for time series-specific manipulation. Also, it has support for many more frequencies, though less customizable by the user (so 5-minutely data is easier to do with pandas for example).

We are aiming to merge these libraries together in the near future.

Progress:

• It has a collection of moving window statistics implemented in Bottleneck
• Outstanding issues
Summarising, Pandas offers superior functionality due to its combination with the `pandas.DataFrame`. An introduction for former users of `scikits.timeseries` is provided in the migration guide.
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 3rd party libraries as they relate to pandas. In offering 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 or harder to use (you may have to be the judge of this given side-by-side code comparisons)

As I do not have an encyclopedic knowledge of R packages, feel free to suggest additional CRAN packages to add to this list. This is also here to offer a big of a translation guide for users of these R packages.

20.1 data.frame

20.2 zoo

20.3 xts

20.4 plyr

20.5 reshape / reshape2
21.1 General functions

21.1.1 Data manipulations

**pivot_table**(data[, values, rows, cols, ...])  
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
- **rows**: list of column names or arrays to group on
  Keys to group on the x-axis of the pivot table
- **cols**: list of column names or arrays to group on
  Keys to group on the y-axis of the pivot table
- **aggfunc**: function, default numpy.mean, or list of functions
  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)
- **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)

*Returns*  
**table**: DataFrame
Examples

```python
>>> 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', rows=['A', 'B'],
                        cols=['C'], aggfunc=np.sum)
```

```
>>> table
     small  large
  foo one  1  4
     two  6  NaN
  bar one  5  4
     two  6  7
```

**merge** *(left, right[, how, on, left_on, ...])*  
Merge DataFrame objects by performing a database-style join operation by columns or indexes.

**concat** *(objs[, axis, join, join_axes, ...])*  
Concatenate pandas objects along a particular axis with optional set logic along the other axes.

**pandas.tools.merge.merge**

pandas.tools.merge.merge *(left, right[, how, on, left_on, ...])*  
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 (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)
  - **on**: label or list
    - Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.
  - **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_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

**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

**Returns**  
merged: DataFrame

**Examples**

```python
>>> A
  lkey  value
0  foo    1
1  bar    2
2  baz    3
3  foo    4

>>> B
  rkey  value
0  foo    5
1  bar    6
2  qux    7
3  bar    8

>>> merge(A, B, left_on='lkey', right_on='rkey', how='outer')
  lkey  value_x  rkey  value_y
0  bar    2    bar    6
1  bar    2    bar    8
2  baz    3    NaN    NaN
3  foo    1    foo    5
4  NaN    NaN    qux    7
```

**pandas.tools.merge.concat**

pandas.tools.merge.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False, keys=None, levels=None, names=None, verify_integrity=False)

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: list or dict 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 an Exception 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)

join_axes : list of Index objects

Specific indexes to use for the other n - 1 axes instead of performing inner/outer set logic

verify_integrity : boolean, default False

Check whether the new concatenated axis contains duplicates. This can be very expensive relative to the actual data concatenation

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

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 the index values on the other axes are still respected in the join.

Returns concatenated : type of objects

Notes

The keys, levels, and names arguments are all optional

21.1.2 Pickling

load(path) Load pickled pandas object (or any other pickled object) from the specified file path

save(obj, path) Pickle (serialize) object to input file path

pandas.core.common.load

pandas.core.common.load(path)

Load pickled pandas object (or any other pickled object) from the specified file path

Parameters path : string

File path

Returns unpickled : type of object stored in file
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

21.1.3 File IO

**read_table**

```python
read_table(filepath_or_buffer[, sep, ...])
```
Read general delimited file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

**Parameters**

- **filepath_or_buffer**: string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, and file. For file URLs, a host is expected. For instance, a local file could be file://localhost/path/to/table.csv
- **sep**: string, default t (tab-stop)
  Delimiter to use. Regular expressions are accepted.
- **lineterminator**: string (length 1), default None
  Character to break file into lines. Only valid with C parser
- **doublequote**: boolean, default False
  Force quoting at all times even if contents look unambiguous
- **escapechar**: string
  Character to escape quoted strings
- **quotechar**: string
  Character to use for quoting
- **skipinitialspace**: boolean, default False
  Skip spaces after delimiter
- **skiprows**: integer or list of integers
  Specify the rows to skip
- **skipfooter**: integer or list of integers
  Specify the number of footer lines to skip
- **na_values**: string or list of strings
  NaN values to convert
- **true_values**: string or list of strings
  True values to convert
- **false_values**: string or list of strings
  False values to convert
- **converters**: dictionary
  Mapping argument names to custom converters
- **dtype**: dictionary
  Select or convert data types
- **usecols**: list of strings
  Columns to read
- **names**: list of strings
  Column names
- **index_col**: integer or string
  Index column
- **skipfooter**: integer or list of integers
  Specify the number of footer lines to skip
- **thousands**: string
  Character used for Thousands separator
- **comment**: string
  Comment pattern
- **dequote**: function
  Accepts a string or a multi-string and returns a list
- **deparse_columns**: function
  String -> list
- **chunksize**: integer
  Number of rows to read at a time
- **nrows**: integer
  Number of rows to read
- **iterator**: boolean
  Iterator is returned

Read general delimited file into DataFrame

21.1. General functions
escapechar : string

dtype : Type name or dict of column -> type

    Data type for data or columns. E.g. {'a': np.float64, 'b': np.int32}

compression : {'gzip', 'bz2', None}, default None

    For on-the-fly decompression of on-disk data

dialect : string or csv.Dialect instance, default None

    If None defaults to Excel dialect. Ignored if sep longer than 1 char See csv.Dialect
documentation for more details

header : int, default 0 if names parameter not specified, otherwise None

    Row to use for the column labels of the parsed DataFrame. Specify None if there is no
    header row.

skiprows : list-like or integer

    Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file

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)

names : array-like

    List of column names to use. If file contains no header row, then you should explicitly
    pass header=None

prefix : string or None (default)

    Prefix to add to column numbers when no header, e.g ‘X’ for X0, X1, ...

na_values : list-like or dict, default None

    Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA
    values

true_values : list

    Values to consider as True

false_values : list

    Values to consider as False

keep_default_na : bool, default True

    If na_values are specified and keep_default_na is False the default NaN values are over-
    ridden, otherwise they’re appended to

parse_dates : boolean, list of ints or names, list of lists, or dict

    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. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’

keep_date_col : boolean, default False

    If True and parse_dates specifies combining multiple columns then keep the original
    columns.
**date_parser** : function

Function to use for converting dates to strings. Defaults to dateutil.parser

**dayfirst** : boolean, default False

DD/MM format dates, international and European format

**thousands** : str, default None

Thousands separator

**comment** : str, default None

Indicates remainder of line should not be parsed. Does not support line commenting (will return empty line)

**decimal** : str, default ‘.’

Character to recognize as decimal point. E.g. use ‘,’ for European data

**nrows** : int, default None

Number of rows of file to read. Useful for reading pieces of large files

**iterator** : boolean, default False

Return TextParser object

**chunksize** : int, default None

Return TextParser object for iteration

**skipfooter** : int, default 0

Number of line at bottom of file to skip

**converters** : dict. optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**verbose** : boolean, default False

Indicate number of NA values placed in non-numeric columns

**delimiter** : string, default None

Alternative argument name for sep. Regular expressions are accepted.

**encoding** : string, default None

Encoding to use for UTF when reading/writing (ex. ‘utf-8’)

**squeeze** : boolean, default False

If the parsed data only contains one column then return a Series

**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

**Returns result** : DataFrame or TextParser
Read CSV (comma-separated) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

**Parameters**

- `filepath_or_buffer` : string or file handle / StringIO. The string could be a URL. Valid URL schemes include http, ftp, and file. For file URLs, a host is expected. For instance, a local file could be file://localhost/path/to/table.csv

- `sep` : string, default `','`
  Delimiter to use. If `sep` is None, will try to automatically determine this. Regular expressions are accepted.

- `lineterminator` : string (length 1), default None
  Character to break file into lines. Only valid with C parser

- `quotechar` : string

- `quoting` : string

- `skipinitialspace` : boolean, default False
  Skip spaces after delimiter

- `escapechar` : string

- `dtype` : Type name or dict of column -> type
  Data type for data or columns. E.g. `{‘a’: np.float64, ‘b’: np.int32}`

- `compression` : {‘gzip’, ‘bz2’, None}, default None
  For on-the-fly decompression of on-disk data

- `dialect` : string or csv.Dialect instance, default None
  If None defaults to Excel dialect. Ignored if `sep` longer than 1 char See csv.Dialect documentation for more details

- `header` : int, default 0 if names parameter not specified, otherwise None
  Row to use for the column labels of the parsed DataFrame. Specify None if there is no header row.

- `skiprows` : list-like or integer
  Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file

- `skipinitialspace` : boolean, default False
  Skip spaces after delimiter

- `escapechar` : string

- `dtype` : string

- `compression` : {‘gzip’, ‘bz2’, None}, default None
  For on-the-fly decompression of on-disk data

- `dialect` : string or csv.Dialect instance, default None
  If None defaults to Excel dialect. Ignored if `sep` longer than 1 char See csv.Dialect documentation for more details

- `header` : int, default 0 if names parameter not specified, otherwise None
  Row to use for the column labels of the parsed DataFrame. Specify None if there is no header row.

- `skiprows` : list-like or integer
  Row numbers to skip (0-indexed) or number of rows to skip (int) at the start of the file
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)

names : array-like

List of column names to use. If file contains no header row, then you should explicitly
pass header=None

prefix : string or None (default)

Prefix to add to column numbers when no header, e.g. ‘X’ for X0, X1, ...

na_values : list-like or dict, default None

Additional strings to recognize as NA/NaN. If dict passed, specific per-column NA
values

true_values : list

Values to consider as True

false_values : list

Values to consider as False

keep_default_na : bool, default True

If na_values are specified and keep_default_na is False the default NaN values are over-
ridden, otherwise they’re appended to

parse_dates : boolean, list of ints or names, list of lists, or dict

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. {'foo' : [1, 3]} -> parse columns 1, 3 as date and call result ‘foo’

keep_date_col : boolean, default False

If True and parse_dates specifies combining multiple columns then keep the original
columns.

date_parser : function

Function to use for converting dates to strings. Defaults to dateutil.parser
dayfirst : boolean, default False

DD/MM format dates, international and European format

thousands : str, default None

Thousands separator

column : str, default None

Indicates remainder of line should not be parsed Does not support line commenting
(will return empty line)
decimal : str, default ‘.’

Character to recognize as decimal point. E.g. use ‘,’ for European data

nrows : int, default None

21.1. General functions
Number of rows of file to read. Useful for reading pieces of large files

**iterator**: boolean, default False

Return TextParser object

**chunksize**: int, default None

Return TextParser object for iteration

**skipfooter**: int, default 0

Number of line at bottom of file to skip

**converters**: dict. optional

Dict of functions for converting values in certain columns. Keys can either be integers or column labels

**verbose**: boolean, default False

Indicate number of NA values placed in non-numeric columns

**delimiter**: string, default None

Alternative argument name for sep. Regular expressions are accepted.

**encoding**: string, default None

Encoding to use for UTF when reading/writing (ex. ‘utf-8’)

**squeeze**: boolean, default False

If the parsed data only contains one column then return a Series

**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

Returns **result**: DataFrame or TextParser

pandas.io.parsers.ExcelFile.parse

**ExcelFile.parse**(sheetname, header=0, skiprows=None, skip_footer=0, index_col=None, parse_cols=None, parse_dates=False, date_parser=None, na_values=None, thousands=None, chunksize=None, **kwds)

Read Excel table into DataFrame

**Parameters**

**sheetname**: string

Name of Excel sheet

**header**: int, default 0

Row to use for the column labels of the parsed DataFrame

**skiprows**: list-like

Rows to skip at the beginning (0-indexed)

**skip_footer**: int, default 0

Rows at the end to skip (0-indexed)

**index_col**: int, default None
Column to use as the row labels of the DataFrame. Pass None if there is no such column

**parse_cols**: 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 column names and
column ranges (e.g. “A:E” or “A,C,E:F”)

**na_values**: list-like, default None

List of additional strings to recognize as NA/NaN

**Returns**  **parsed**: DataFrame

### 21.1.4 HDFStore: PyTables (HDF5)

**HDFStore.put(key, value[, table, append])**  Store object in HDFStore

**HDFStore.get(key)**  Retrieve pandas object stored in file

#### pandas.io.pytables.HDFStore.put

**HDFStore.put**(key, value, table=None, append=False, **kwargs)

Store object in HDFStore

**Parameters**  **key**: object

**value**: {Series, DataFrame, Panel}

**table**: boolean, default False

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

For table data structures, append the input data to the existing table

#### pandas.io.pytables.HDFStore.get

**HDFStore.get**(key)

Retrieve pandas object stored in file

**Parameters**  **key**: object

**Returns**  **obj**: type of object stored in file

### 21.1.5 Standard moving window functions

- **rolling_count**: arg, window[, freq, center, ...])  Rolling count of number of non-NaN observations inside provided window.
- **rolling_sum**: arg, window[, min_periods, ...])  Moving sum
- **rolling_mean**: arg, window[, min_periods, ...])  Moving mean
- **rolling_median**: arg, window[, min_periods, ...])  O(N log(window)) implementation using skip list
- **rolling_var**: arg, window[, min_periods, ...])  Unbiased moving variance

Continued on next page
Table 21.6 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rolling_std</code></td>
<td>Unbiased moving standard deviation</td>
</tr>
<tr>
<td><code>rolling_corr</code></td>
<td>Moving sample correlation</td>
</tr>
<tr>
<td><code>rolling_cov</code></td>
<td>Unbiased moving covariance</td>
</tr>
<tr>
<td><code>rolling_skew</code></td>
<td>Unbiased moving skewness</td>
</tr>
<tr>
<td><code>rolling_kurt</code></td>
<td>Unbiased moving kurtosis</td>
</tr>
<tr>
<td><code>rolling_apply</code></td>
<td>Generic moving function application</td>
</tr>
<tr>
<td><code>rolling_quantile</code></td>
<td>Moving quantile</td>
</tr>
</tbody>
</table>

**pandas.stats.moments.rolling_count**

`pandas.stats.moments.rolling_count` (arg, window, freq=None, center=False, time_rule=None)

Rolling count of number of non-NaN observations inside provided window.

Parameters
- **arg**: DataFrame or numpy ndarray-like
- **window**: Number of observations used for calculating statistic
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic
- **center**: boolean, default False
  - Whether the label should correspond with center of window

Returns **rolling_count**: type of caller

**pandas.stats.moments.rolling_sum**

`pandas.stats.moments.rolling_sum` (arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Moving sum

Parameters
- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns **y**: type of input argument

**pandas.stats.moments.rolling_mean**

`pandas.stats.moments.rolling_mean` (arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Moving mean

Parameters
- **arg**: Series, DataFrame
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns y : type of input argument

pandas.stats.moments.rolling_median

pandas.stats.moments.rolling_median(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

O(N log(window)) implementation using skip list

Moving median

Parameters
arg : Series, DataFrame
window : Number of observations used for calculating statistic
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns y : type of input argument

pandas.stats.moments.rolling_var

pandas.stats.moments.rolling_var(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Unbiased moving variance

Parameters
arg : Series, DataFrame
window : Number of observations used for calculating statistic
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns y : type of input argument

pandas.stats.moments.rolling_std

pandas.stats.moments.rolling_std(arg, window, min_periods=None, freq=None, center=False, time_rule=None, **kwargs)

Unbiased moving standard deviation

Parameters
arg : Series, DataFrame
window : Number of observations used for calculating statistic
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

Moving sample correlation

Parameters
- **arg1**: Series, DataFrame, or ndarray
- **arg2**: Series, DataFrame, or ndarray
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns
- **y**: type depends on inputs
  - DataFrame / DataFrame -> DataFrame (matches on columns)
  - DataFrame / Series -> Computes result for each column
  - Series / Series -> Series

Unbiased moving covariance

Parameters
- **arg1**: Series, DataFrame, or ndarray
- **arg2**: Series, DataFrame, or ndarray
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns
- **y**: type depends on inputs
  - DataFrame / DataFrame -> DataFrame (matches on columns)
  - DataFrame / Series -> Computes result for each column
  - Series / Series -> Series

Unbiased moving skewness

Parameters
- **arg**: Series, DataFrame, or ndarray
- **window**: Number of observations used for calculating statistic
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic time_rule is a legacy alias for freq

Returns
- **y**: type depends on inputs
  - DataFrame / DataFrame -> DataFrame (matches on columns)
  - DataFrame / Series -> Computes result for each column
  - Series / Series -> Series
Parameters

**arg**: Series, DataFrame

  **window**: Number of observations used for calculating statistic

  **min_periods**: int

  Minimum number of observations in window required to have a value

  **freq**: None or string alias / date offset object, default=None

  Frequency to conform to before computing statistic

  **time_rule** is a legacy alias for freq

Returns

**y**: type of input argument

---

**pandas.stats.moments.rolling_kurt**

\[ \text{roll}_n \text{mean} \]

Unbiased moving kurtosis

Parameters

**arg**: Series, DataFrame

  **window**: Number of observations used for calculating statistic

  **min_periods**: int

  Minimum number of observations in window required to have a value

  **freq**: None or string alias / date offset object, default=None

  Frequency to conform to before computing statistic

  **time_rule** is a legacy alias for freq

Returns

**y**: type of input argument

---

**pandas.stats.moments.rolling_apply**

\[ \text{func}_n \]

Generic moving function application

Parameters

**arg**: Series, DataFrame

  **window**: Number of observations used for calculating statistic

  **func**: function

  Must produce a single value from an ndarray input

  **min_periods**: int

  Minimum number of observations in window required to have a value

  **freq**: None or string alias / date offset object, default=None

  Frequency to conform to before computing statistic

  **center**: boolean, default False

  Whether the label should correspond with center of window

Returns

**y**: type of input argument

---

21.1. General functions
pandas.stats.moments.rolling_quantile

Moving quantile

**Parameters**
- `arg` : Series, DataFrame
- `window` : Number of observations used for calculating statistic
- `quantile` : 0 <= quantile <= 1
- `min_periods` : int
  Minimum number of observations in window required to have a value
- `freq` : None or string alias / date offset object, default=None
  Frequency to conform to before computing statistic
- `center` : boolean, default False
  Whether the label should correspond with center of window

**Returns**
- `y` : type of input argument

### 21.1.6 Standard expanding window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expanding_count</code></td>
<td>Expanding count of number of non-NaN observations.</td>
</tr>
<tr>
<td><code>expanding_sum</code></td>
<td>Expanding sum</td>
</tr>
<tr>
<td><code>expanding_mean</code></td>
<td>Expanding mean</td>
</tr>
<tr>
<td><code>expanding_median</code></td>
<td>O(N log(window)) implementation using skip list</td>
</tr>
<tr>
<td><code>expanding_var</code></td>
<td>Unbiased expanding variance</td>
</tr>
<tr>
<td><code>expanding_std</code></td>
<td>Unbiased expanding standard deviation</td>
</tr>
<tr>
<td><code>expanding_corr</code></td>
<td>Expanding sample correlation</td>
</tr>
<tr>
<td><code>expanding_cov</code></td>
<td>Unbiased expanding covariance</td>
</tr>
<tr>
<td><code>expanding_skew</code></td>
<td>Unbiased expanding skewness</td>
</tr>
<tr>
<td><code>expanding_kurt</code></td>
<td>Unbiased expanding kurtosis</td>
</tr>
<tr>
<td><code>expanding_apply</code></td>
<td>Generic expanding function application</td>
</tr>
<tr>
<td><code>expanding_quantile</code></td>
<td>Expanding quantile</td>
</tr>
</tbody>
</table>

pandas.stats.moments.expanding_count

Expanding count of number of non-NaN observations.

**Parameters**
- `arg` : DataFrame or numpy ndarray-like
- `freq` : None or string alias / date offset object, default=None
  Frequency to conform to before computing statistic
- `center` : boolean, default False
  Whether the label should correspond with center of window

**Returns**
- `expanding_count` : type of caller
pandas.stats.moments.expanding_sum

Expanding sum

Parameters
- **arg**: Series, DataFrame
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

Returns
- **y**: type of input argument

pandas.stats.moments.expanding_mean

Expanding mean

Parameters
- **arg**: Series, DataFrame
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

Returns
- **y**: type of input argument

pandas.stats.moments.expanding_median

Expanding median

Parameters
- **arg**: Series, DataFrame
- **min_periods**: int
  - Minimum number of observations in window required to have a value
- **freq**: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

Returns
- **y**: type of input argument

pandas.stats.moments.expanding_var

Unbiased expanding variance
**Parameters**

- `arg`: Series, DataFrame
- `min_periods`: int
  - Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

**Returns**

- `y`: type of input argument

---

**pandas.stats.moments.expanding_std**

Unbiased expanding standard deviation

- `arg`: Series, DataFrame
- `min_periods`: int
  - Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

**Returns**

- `y`: type of input argument

---

**pandas.stats.moments.expanding_corr**

Expanding sample correlation

- `arg1`: Series, DataFrame, or ndarray
- `arg2`: Series, DataFrame, or ndarray
- `min_periods`: int
  - Minimum number of observations in window required to have a value
- `freq`: None or string alias / date offset object, default=None
  - Frequency to conform to before computing statistic

**Returns**

- `y`: type depends on inputs
  - DataFrame / DataFrame -> DataFrame (matches on columns)
  - DataFrame / Series -> Computes result for each column
  - Series / Series -> Series

---

**pandas.stats.moments.expanding_cov**

Unbiased expanding covariance

- `arg1`: Series, DataFrame, or ndarray
- `arg2`: Series, DataFrame, or ndarray
- `min_periods`: int
Minimum number of observations in window required to have a value

**freq** : None or string alias / date offset object, default=None

Frequency to conform to before computing statistic

**Returns**

<table>
<thead>
<tr>
<th>type</th>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame / DataFrame -&gt; DataFrame</td>
<td>matches on columns</td>
<td>Computes result for each column</td>
</tr>
<tr>
<td>DataFrame / Series -&gt; Series</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**pandas.stats.moments.expanding_skew**

Unbiased expanding skewness

**Parameters**

<table>
<thead>
<tr>
<th>arg</th>
<th>Series, DataFrame</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>min_periods</strong></td>
<td>int</td>
</tr>
<tr>
<td>Minimum number of observations in window required to have a value</td>
<td></td>
</tr>
<tr>
<td><strong>freq</strong></td>
<td>None or string alias / date offset object, default=None</td>
</tr>
<tr>
<td>Frequency to conform to before computing statistic</td>
<td></td>
</tr>
</tbody>
</table>

**Returns**

| y | type of input argument |

**pandas.stats.moments.expanding_kurt**

Unbiased expanding kurtosis

**Parameters**

<table>
<thead>
<tr>
<th>arg</th>
<th>Series, DataFrame</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>min_periods</strong></td>
<td>int</td>
</tr>
<tr>
<td>Minimum number of observations in window required to have a value</td>
<td></td>
</tr>
<tr>
<td><strong>freq</strong></td>
<td>None or string alias / date offset object, default=None</td>
</tr>
<tr>
<td>Frequency to conform to before computing statistic</td>
<td></td>
</tr>
</tbody>
</table>

**Returns**

| y | type of input argument |

**pandas.stats.moments.expanding_apply**

Generic expanding function application

**Parameters**

<table>
<thead>
<tr>
<th>arg</th>
<th>Series, DataFrame</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>func</strong></td>
<td>function</td>
</tr>
<tr>
<td>Must produce a single value from an ndarray input</td>
<td></td>
</tr>
<tr>
<td><strong>min_periods</strong></td>
<td>int</td>
</tr>
<tr>
<td>Minimum number of observations in window required to have a value</td>
<td></td>
</tr>
</tbody>
</table>

21.1. General functions
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic

center : boolean, default False
Whether the label should correspond with center of window

Returns y : type of input argument

**pandas.stats.moments.expanding_quantile**

Expanding quantile

Parameters arg : Series, DataFrame
quantile : 0 <= quantile <= 1
min_periods : int
Minimum number of observations in window required to have a value
freq : None or string alias / date offset object, default=None
Frequency to conform to before computing statistic
center : boolean, default False
Whether the label should correspond with center of window

Returns y : type of input argument

### 21.1.7 Exponentially-weighted moving window functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ewma(arg[, com, span, min_periods, freq, ...])</code></td>
<td>Exponentially-weighted moving average</td>
</tr>
<tr>
<td><code>ewmatstd(arg[, com, span, min_periods, bias, ...])</code></td>
<td>Exponentially-weighted moving std</td>
</tr>
<tr>
<td><code>ewmvar(arg[, com, span, min_periods, bias, ...])</code></td>
<td>Exponentially-weighted moving variance</td>
</tr>
<tr>
<td><code>ewmcorr(arg1, arg2[, com, span, ...])</code></td>
<td>Exponentially-weighted moving correlation</td>
</tr>
<tr>
<td><code>ewmxcov(arg1, arg2[, com, span, min_periods, ...])</code></td>
<td>Exponentially-weighted moving covariance</td>
</tr>
</tbody>
</table>

**pandas.stats.moments.ewma**

Exponentially-weighted moving average

Parameters arg : Series, DataFrame
com : float, optional
Center of mass: alpha = com / (1 + com),
span : float, optional
Specify decay in terms of span, alpha = 2 / (span + 1)
min_periods : int, default 0
Number of observations in sample to require (only affects beginning)
freq : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic time_rule is a legacy alias for freq

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)

Returns  y : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter s, we have have that the decay parameter alpha is
related to the span as $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

where c is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$
So a “20-day EWMA” would have center 9.5.

pandas.stats.moments.ewmstd

pandas.stats.moments.ewmstd(arg, com=None, span=None, min_periods=0, bias=False, 
                           time_rule=None)

Exponentially-weighted moving std

Parameters  arg : Series, DataFrame

com : float, optional
    Center of mass: alpha = com / (1 + com),

span : float, optional
    Specify decay in terms of span, alpha = 2 / (span + 1)

min_periods : int, default 0
    Number of observations in sample to require (only affects beginning)

freq : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic time_rule is a legacy alias for freq

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)

bias : boolean, default False
    Use a standard estimation bias correction

Returns  y : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter s, we have have that the decay parameter alpha is
related to the span as $\alpha = 1 - 2/(s + 1) = c/(1 + c)$
where $c$ is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$.
So a “20-day EWMA” would have center 9.5.

**pandas.stats.moments.ewmvar**

```
pandas.stats.moments.ewmvar(arg, com=None, span=None, min_periods=0, bias=False, freq=None, time_rule=None)
```

Exponentially-weighted moving variance

**Parameters**

- **arg** : Series, DataFrame
  - `com` : float, optional
    - Center of mass: $\alpha = \frac{com}{1 + com}$,
  - `span` : float, optional
    - Specify decay in terms of span, $\alpha = \frac{2}{span + 1}$
  - `min_periods` : int, default 0
    - Number of observations in sample to require (only affects beginning)
  - `freq` : None or string alias / date offset object, default=None
    - Frequency to conform to before computing statistic, time_rule is a legacy alias for freq
  - `bias` : boolean, default False
    - Use a standard estimation bias correction

**Returns**

- **$y$** : type of input argument

**Notes**

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter $s$, we have have that the decay parameter alpha is related to the span as $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

where $c$ is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$
So a “20-day EWMA” would have center 9.5.

**pandas.stats.moments.ewmcorr**

```
pandas.stats.moments.ewmcorr(arg1, arg2, com=None, span=None, min_periods=0, freq=None, time_rule=None)
```

Exponentially-weighted moving correlation

**Parameters**

- **arg1** : Series, DataFrame, or ndarray
  - `arg2` : Series, DataFrame, or ndarray
  - `com` : float, optional
    - Center of mass: $\alpha = \frac{com}{1 + com}$,
span : float, optional
    Specify decay in terms of span, alpha = 2 / (span + 1)

min_periods : int, default 0
    Number of observations in sample to require (only affects beginning)

freq : None or string alias / date offset object, default=None
    Frequency to conform to before computing statistic time_rule is a legacy alias for freq

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)

Returns  y : type of input argument

Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter s, we have have that the decay parameter alpha is
related to the span as α = 1 − 2/(s + 1) = c/(1 + c)

where c is the center of mass. Given a span, the associated center of mass is c = (s − 1)/2

So a “20-day EWMA” would have center 9.5.

pandas.stats.moments.ewmcov

pandas.stats.moments.ewmcov(arg1, arg2, com=None, span=None, min_periods=0, bias=False,
                             freq=None, time_rule=None)

Exponentially-weighted moving covariance

Parameters  arg1 : Series, DataFrame, or ndarray
            arg2 : Series, DataFrame, or ndarray
            com : float. optional
                Center of mass: alpha = com / (1 + com),
            span : float, optional
                Specify decay in terms of span, alpha = 2 / (span + 1)
            min_periods : int, default 0
                Number of observations in sample to require (only affects beginning)
            freq : None or string alias / date offset object, default=None
                Frequency to conform to before computing statistic time_rule is a legacy alias for freq
            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)

Returns  y : type of input argument

21.1. General functions
Notes

Either center of mass or span must be specified

EWMA is sometimes specified using a “span” parameter $s$, we have have that the decay parameter $\alpha$ is related to the span as $\alpha = 1 - 2/(s + 1) = c/(1 + c)$

where $c$ is the center of mass. Given a span, the associated center of mass is $c = (s - 1)/2$

So a “20-day EWMA” would have center 9.5.

21.2 Series

21.2.1 Attributes and underlying data

Axes

- `index`: axis labels

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.values</code></td>
<td>Return Series as ndarray</td>
</tr>
<tr>
<td><code>Series.dtype</code></td>
<td>Data-type of the array’s elements.</td>
</tr>
<tr>
<td><code>Series.isnull(obj)</code></td>
<td>Detect missing values (NaN in numeric arrays, None/NaN in object arrays)</td>
</tr>
<tr>
<td><code>Series.notnull(obj)</code></td>
<td>Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.</td>
</tr>
</tbody>
</table>

**pandas.Series.values**

`Series.values`

Return Series as ndarray

Returns `arr`: numpy.ndarray

**pandas.Series.dtype**

`Series.dtype`

Data-type of the array’s elements.

Parameters: None

Returns `d`: numpy dtype object

See Also:

`numpy.dtype`

Examples

```python
>>> x
array([[0, 1],
       [2, 3]])
```

```python
>>> x.dtype
dtype('int32')
```

```python
>>> type(x.dtype)
type 'numpy.dtype'
```
pandas.Series.isnull

Series.isnull(obj)
Detect missing values (NaN in numeric arrays, None/NaN in object arrays)

Parameters  arr: ndarray or object value :

Returns  boolean ndarray or boolean :

pandas.Series.notnull

Series.notnull(obj)
Replacement for numpy.isfinite / -numpy.isnan which is suitable for use on object arrays.

Parameters  arr: ndarray or object value :

Returns  boolean ndarray or boolean :

21.2.2 Conversion / Constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.<strong>init</strong>([data, index, dtype, name, copy])</td>
<td>One-dimensional ndarray with axis labels (including time series).</td>
</tr>
<tr>
<td>Series.astype(dtype)</td>
<td>See numpy.ndarray.astype</td>
</tr>
<tr>
<td>Series.copy([order])</td>
<td>Return new Series with copy of underlying values</td>
</tr>
</tbody>
</table>

pandas.Series.__init__

Series.__init__(data=None, index=None, dtype=None, name=None, copy=False)
One-dimensional ndarray with axis labels (including time series). Labels need not be unique but must be any 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

index : array-like or Index (1d)

Values must be unique and hashable, same length as data. Index object (or other iterable of same length as data) Will default to np.arange(len(data)) 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

pandas.Series.astype

Series.astype(dtype)
See numpy.ndarray.astype
**pandas.Series.copy**

Series.copy(order='C')

Return new Series with copy of underlying values

Returns cp : Series

### 21.2.3 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.get(label[, default])</td>
<td>Returns value occupying requested label, default to specified missing value if not present.</td>
</tr>
<tr>
<td>Series.ix</td>
<td></td>
</tr>
<tr>
<td>Series.<strong>iter</strong>()</td>
<td></td>
</tr>
<tr>
<td>Series.iteritems()</td>
<td>Lazily iterate over (index, value) tuples</td>
</tr>
</tbody>
</table>

**pandas.Series.get**

Series.get(label, default=None)

Returns value occupying requested label, default to specified missing value if not present. Analogous to dict.get

Parameters

- **label** : object
  - Label value looking for
- **default** : object, optional
  - Value to return if label not in index

Returns y : scalar

**pandas.Series.ix**

Series.ix

**pandas.Series.__iter__**

Series.__iter__()

**pandas.Series.iteritems**

Series.iteritems()

Lazily iterate over (index, value) tuples

### 21.2.4 Binary operator functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.add(other[, level, fill_value])</td>
<td>Binary operator add with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.div(other[, level, fill_value])</td>
<td>Binary operator divide with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.mul(other[, level, fill_value])</td>
<td>Binary operator multiply with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.sub(other[, level, fill_value])</td>
<td>Binary operator subtract with support to substitute a fill_value for missing data</td>
</tr>
<tr>
<td>Series.combine(other, func[, fill_value])</td>
<td>Perform elementwise binary operation on two Series using given function</td>
</tr>
</tbody>
</table>

Continued on next page
pandas.Series.add

Series.add(other, level=None, fill_value=None)

Binary operator add 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 missing (NaN) values with this value. If both Series are 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

pandas.Series.div

Series.div(other, level=None, fill_value=None)

Binary operator divide 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 missing (NaN) values with this value. If both Series are 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

pandas.Series.mul

Series.mul(other, level=None, fill_value=None)

Binary operator multiply 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 missing (NaN) values with this value. If both Series are 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

---

Table 21.12 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.combine_first(other)</td>
<td>Combine Series values, choosing the calling Series's values</td>
</tr>
<tr>
<td>Series.round([decimals, out])</td>
<td>Return a with each element rounded to the given number of decimals.</td>
</tr>
</tbody>
</table>
pandas.Series.sub

Series.sub(other, level=None, fill_value=None)
Binary operator subtract 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 missing (NaN) values with this value. If both Series are 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

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
fill_value: scalar value

Returns
result: Series

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
y: Series

pandas.Series.round

Series.round(decimals=0, out=None)
Return a with each element rounded to the given number of decimals.

Refer to numpy.around for full documentation.

See Also:

numpy.around equivalent function

Continued on next page
21.2.5 Function application, GroupBy

<table>
<thead>
<tr>
<th>pandas.Series.apply</th>
<th>pandas.Series.map</th>
<th>pandas.Series.groupby</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>Map values of Series using input correspondence (which can be a dict, Series, or function).</td>
<td>Group series using mapper (dict or key function, apply given function).</td>
</tr>
</tbody>
</table>

**Series.apply**

```
Series.apply(func[, convert_dtypes, args])
```

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_dtypes: boolean, default True
  - Try to find better dtype for elementwise function results. If False, leave as dtype=object.

**Returns**

- y: Series or DataFrame if func returns a Series

**See Also:**

Series.map For element-wise operations

**Series.map**

```
Series.map(arg[, na_action])
```

Map values of Series using input correspondence (which can be a dict, Series, or function).

**Parameters**

- **arg**: function, dict, or Series
  - na_action: {None, ‘ignore’}
  - If ‘ignore’, propagate NA values

**Returns**

- y: Series
  - same index as caller

**Examples**

```python
>>> x
one 1
two 2
three 3

>>> y
1 foo
2 bar
3 baz
```
```python
>>> x.map(y)
one  foo
two  bar
three baz
```

### pandas.Series.groupby

**Series.groupby** *(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True)*

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 / list of functions, dict, Series, or tuple / list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups
- **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
- **group_keys**: boolean, default True
  - When calling apply, add group keys to index to identify pieces

**Returns**
- **GroupBy object**

**Examples**

- # DataFrame result >>> data.groupby(func, axis=0).mean()
- # DataFrame result >>> data.groupby([‘col1’, ‘col2’])[‘col3’].mean()
- # DataFrame with hierarchical index >>> data.groupby([‘col1’, ‘col2’]).mean()

## 21.2.6 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.abs()</td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td>Series.any([axis, out])</td>
<td>Returns True if any of the elements of a evaluate to True.</td>
</tr>
<tr>
<td>Series.autocorr()</td>
<td>Lag-1 autocorrelation</td>
</tr>
<tr>
<td>Series.between(left, right[, inclusive])</td>
<td>Return boolean Series equivalent to left &lt;= series &lt;= right. NA values</td>
</tr>
<tr>
<td>Series.clip([lower, upper, out])</td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td>Series.clip_lower([threshold])</td>
<td>Return copy of series with values below given value truncated</td>
</tr>
<tr>
<td>Series.clip_upper([threshold])</td>
<td>Return copy of series with values above given value truncated</td>
</tr>
<tr>
<td>Series.corr(other[, method, min_periods])</td>
<td>Compute correlation with other Series, excluding missing values</td>
</tr>
</tbody>
</table>
Table 21.14 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.count([level])</td>
<td>Return number of non-NA/null observations in the Series</td>
</tr>
<tr>
<td>Series.cov(other[, min_periods])</td>
<td>Compute covariance with Series, excluding missing values</td>
</tr>
<tr>
<td>Series.cummax([axis, dtype, out, skipna])</td>
<td>Cumulative max of values.</td>
</tr>
<tr>
<td>Series.cummin([axis, dtype, out, skipna])</td>
<td>Cumulative min of values.</td>
</tr>
<tr>
<td>Series.cumprod([axis, dtype, out, skipna])</td>
<td>Cumulative product of values.</td>
</tr>
<tr>
<td>Series.cumsum([axis, dtype, out, skipna])</td>
<td>Cumulative sum of values.</td>
</tr>
<tr>
<td>Series.describe([percentile_width])</td>
<td>Generate various summary statistics of Series, excluding NaN</td>
</tr>
<tr>
<td>Series.diff([periods])</td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td>Series.kurt([skipna, level])</td>
<td>Return unbiased kurtosis of values</td>
</tr>
<tr>
<td>Series.mad([skipna, level])</td>
<td>Return mean absolute deviation of values</td>
</tr>
<tr>
<td>Series.max([axis, out, skipna, level])</td>
<td>Return maximum of values</td>
</tr>
<tr>
<td>Series.mean([axis, dtype, out, skipna, level])</td>
<td>Return mean of values</td>
</tr>
<tr>
<td>Series.median([axis, dtype, out, skipna, level])</td>
<td>Return median of values</td>
</tr>
<tr>
<td>Series.min([axis, out, skipna, level])</td>
<td>Return minimum of values</td>
</tr>
<tr>
<td>Series.nunique()</td>
<td>Return count of unique elements in the Series</td>
</tr>
<tr>
<td>Series.pct_change([periods, fill_method, ...])</td>
<td>Percent change over given number of periods</td>
</tr>
<tr>
<td>Series.prod([axis, dtype, out, skipna, level])</td>
<td>Return product of values</td>
</tr>
<tr>
<td>Series.quantile([q])</td>
<td>Return value at the given quantile, a la scoreatpercentile in</td>
</tr>
<tr>
<td>Series.rank([method, na_option, ascending])</td>
<td>Compute data ranks (1 through n).</td>
</tr>
<tr>
<td>Series.skew([skipna, level])</td>
<td>Return unbiased skewness of values</td>
</tr>
<tr>
<td>Series.std([axis, dtype, out, ddof, skipna, ...])</td>
<td>Return standard deviation of values</td>
</tr>
<tr>
<td>Series.sum([axis, dtype, out, skipna, level])</td>
<td>Return sum of values</td>
</tr>
<tr>
<td>Series.unique()</td>
<td>Return array of unique values in the Series. Significantly faster than</td>
</tr>
<tr>
<td>Series.var([axis, dtype, out, ddof, skipna, ...])</td>
<td>Return variance of values</td>
</tr>
<tr>
<td>Series.value_counts([normalize])</td>
<td>Returns Series containing counts of unique values. The resulting Series</td>
</tr>
</tbody>
</table>

**pandas.Series.abs**

Series.abs()

Return an object with absolute value taken. Only applicable to objects that are all numeric

Returns abs: type of caller:

**pandas.Series.any**

Series.any(axis=None, out=None)

Returns True if any of the elements of a evaluate to True.

Refer to numpy.any for full documentation.

See Also:

numpy.any equivalent function

**pandas.Series.autocorr**

Series.autocorr()

Lag-1 autocorrelation

Returns autocorr: float
pandas.Series.between

Series.between(left, right, inclusive=True)
Return boolean Series equivalent to left <= series <= right. NA values will be treated as False

Parameters:
left : scalar
    Left boundary
right : scalar
    Right boundary

Returns:
is_between : Series

pandas.Series.clip

Series.clip(lower=None, upper=None, out=None)
Trim values at input threshold(s)

Parameters:
lower : float, default None
    upper : float, default None

Returns:
clipped : Series

pandas.Series.clip_lower

Series.clip_lower(threshold)
Return copy of series with values below given value truncated

Returns:
clipped : Series

See Also:
clip

pandas.Series.clip_upper

Series.clip_upper(threshold)
Return copy of series with values above given value truncated

Returns:
clipped : Series

See Also:
clip

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'}
    pearsen : standard correlation coefficient
    kendall : Kendall Tau correlation coefficient
    spearman : Spearman rank correlation
**pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f**

**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, 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=0, dtype=None, out=None, skipna=True)*  
Cumulative max of values. Preserves locations of NaN values

Extra parameters are to preserve ndarray interface.

**Parameters**  
skipna : boolean, default True

Exclude NA/null values

**Returns**  
cummax : Series

**pandas.Series.cummin**

Series**.cummin** *(axis=0, dtype=None, out=None, skipna=True)*  
Cumulative min of values. Preserves locations of NaN values

Extra parameters are to preserve ndarray interface.

**Parameters**  
skipna : boolean, default True

Exclude NA/null values

**Returns**  
cummin : Series
**pandas.Series.cumprod**

`Series.cumprod(axis=0, dtype=None, out=None, skipna=True)`

Cumulative product of values. Preserves locations of NaN values.

Extra parameters are to preserve ndarray interface.

- **Parameters**
  - `skipna`: boolean, default True
    - Exclude NA/null values

- **Returns**
  - `cumprod`: Series

**pandas.Series.cumsum**

`Series.cumsum(axis=0, dtype=None, out=None, skipna=True)`

Cumulative sum of values. Preserves locations of NaN values.

Extra parameters are to preserve ndarray interface.

- **Parameters**
  - `skipna`: boolean, default True
    - Exclude NA/null values

- **Returns**
  - `cumsum`: Series

**pandas.Series.describe**

`Series.describe(percentile_width=50)`

Generate various summary statistics of Series, excluding NaN values. These include: count, mean, std, min, max, and lower%/50%/upper% percentiles.

- **Parameters**
  - `percentile_width`: float, optional
    - Width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

- **Returns**
  - `desc`: Series

**pandas.Series.diff**

`Series.diff(periods=1)`

1st discrete difference of object.

- **Parameters**
  - `periods`: int, default 1
    - Periods to shift for forming difference

- **Returns**
  - `difed`: Series

**pandas.Series.kurt**

`Series.kurt(skipna=True, level=None)`

Return unbiased kurtosis of values NA/null values are excluded.

- **Parameters**
  - `skipna`: boolean, default True
    - Exclude NA/null values
  - `level`: int, default None
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a smaller Series

**Returns**  
**kurt** : float (or Series if level specified)

**pandas.Series.mad**

Series.mad(*skipna=True, level=None*)

Return mean absolute deviation of values NA/null values are excluded

**Parameters**  
**skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a smaller Series

**Returns**  
**mad** : float (or Series if level specified)

**pandas.Series.max**

Series.max(*axis=None, out=None, skipna=True, level=None*)

Return maximum of values NA/null values are excluded

**Parameters**  
**skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a smaller Series

**Returns**  
**max** : float (or Series if level specified)

**pandas.Series.mean**

Series.mean(*axis=0, dtype=None, out=None, skipna=True, level=None*)

Return mean of values NA/null values are excluded

**Parameters**  
**skipna** : boolean, default True

Exclude NA/null values

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a smaller Series

**Extra parameters are to preserve ndarrayinterface.**

**Returns**  
**mean** : float (or Series if level specified)
**pandas.Series.median**

Series.median \((axis=0, dtype=None, out=None, skipna=True, level=None)\)
Return median of values NA/null values are excluded

- **Parameters**
  - **skipna**: boolean, default True
    Exclude NA/null values
  - **level**: int, default None
    If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

- **Returns**
  - **median**: float (or Series if level specified)

**pandas.Series.min**

Series.min \((axis=None, out=None, skipna=True, level=None)\)
Return minimum of values NA/null values are excluded

- **Parameters**
  - **skipna**: boolean, default True
    Exclude NA/null values
  - **level**: int, default None
    If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

- **Returns**
  - **min**: float (or Series if level specified)

**pandas.Series.nunique**

Series.nunique()
Return count of unique elements in the Series

- **Returns**
  - **nunique**: int

**pandas.Series.pct_change**

Series.pct_change \((periods=1, fill_method='pad', limit=None, freq=None, **kwds)\)
Percent change over given number of periods

- **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())

- **Returns**
  - **chg**: Series or DataFrame
pandas.Series.prod

Series.prod(axis=0, dtype=None, out=None, skipna=True, level=None)
Return product of values NA/null values are excluded

Parameters
- skipna : boolean, default True
  Exclude NA/null values
- level : int, default None
  If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
  a smaller Series

Returns
- prod : float (or Series if level specified)

pandas.Series.quantile

Series.quantile(q=0.5)
Return value at the given quantile, a la scoreatpercentile in scipy.stats

Parameters
- q : quantile
  0 <= q <= 1

Returns
- quantile : float

pandas.Series.rank

Series.rank(method='average', na_option='keep', ascending=True)
Compute data ranks (1 through n). Equal values are assigned a rank that is the average of the ranks of those
values

Parameters
- method : {'average', 'min', 'max', 'first'}
  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
- na_option : {'keep'}
  keep: leave NA values where they are
- ascending : boolean, default True
  False for ranks by high (1) to low (N)

Returns
- ranks : Series

pandas.Series.skew

Series.skew(skipna=True, level=None)
Return unbiased skewness of values NA/null values are excluded

Parameters
- skipna : boolean, default True
  Exclude NA/null values
- level : int, default None
  If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
  a smaller Series
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

**Returns**  
*skew*: float (or Series if level specified)

**pandas.Series.std**

Series.*std*(axis=None, dtype=None, out=None, ddof=1, skipna=True, level=None)  
Return standard deviation of values NA/null values are excluded

**Parameters**  
*skipna*: boolean, default True  
Exclude NA/null values

*level*: int, default None  
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns**  
*stdev*: float (or Series if level specified)  
Normalized by N-1 (unbiased estimator).

**pandas.Series.sum**

Series.*sum*(axis=0, dtype=None, out=None, skipna=True, level=None)  
Return sum of values NA/null values are excluded

**Parameters**  
*skipna*: boolean, default True  
Exclude NA/null values

*level*: int, default None  
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Extra parameters are to preserve ndarrayinterface.**

**Returns**  
*sum*: float (or Series if level specified)

**pandas.Series.unique**

Series.*unique*()  
Return array of unique values in the Series. Significantly faster than numpy.unique

**Returns**  
*uniques*: ndarray

**pandas.Series.var**

Series.*var*(axis=None, dtype=None, out=None, ddof=1, skipna=True, level=None)  
Return variance of values NA/null values are excluded

**Parameters**  
*skipna*: boolean, default True  
Exclude NA/null values

*level*: int, default None  
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a smaller Series

**Returns**  
*var*: float (or Series if level specified)
Normalized by N-1 (unbiased estimator).

**pandas.Series.value_counts**

**Series.value_counts** *(normalize=False)*

Returns Series containing counts of unique values. The resulting Series will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values.

- **Parameters**
  - `normalize`: boolean, default False

  If True then the Series returned will contain the relative frequencies of the unique values.

- **Returns**
  - `counts`: Series

**21.2.7 Reindexing / Selection / Label manipulation**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.align</td>
<td>Align two Series object with the specified join method</td>
</tr>
<tr>
<td>Series.drop</td>
<td>Return new object with labels in requested axis removed</td>
</tr>
<tr>
<td>Series.first</td>
<td>Convenience method for subsetting initial periods of time series data</td>
</tr>
<tr>
<td>Series.head</td>
<td>Returns first n rows of Series</td>
</tr>
<tr>
<td>Series.idxmax</td>
<td>Index of first occurrence of maximum of values.</td>
</tr>
<tr>
<td>Series.idxmin</td>
<td>Index of first occurrence of minimum of values.</td>
</tr>
<tr>
<td>Series.isin</td>
<td>Return boolean vector showing whether each element in the Series is in values</td>
</tr>
<tr>
<td>Series.last</td>
<td>Convenience method for subsetting final periods of time series data</td>
</tr>
<tr>
<td>Series.reindex</td>
<td>Conform Series to new index with optional filling logic, placing</td>
</tr>
<tr>
<td>Series.reindex_like</td>
<td>Reindex Series to match index of another Series, optionally with</td>
</tr>
<tr>
<td>Series.rename</td>
<td>Alter Series index using dict or function</td>
</tr>
<tr>
<td>Series.reset_index</td>
<td>Analogous to the DataFrame.reset_index function, see docstring there.</td>
</tr>
<tr>
<td>Series.select</td>
<td>Return data corresponding to axis labels matching criteria</td>
</tr>
<tr>
<td>Series.take</td>
<td>Analogous to ndarray.take, return Series corresponding to requested</td>
</tr>
<tr>
<td>Series.tail</td>
<td>Returns last n rows of Series</td>
</tr>
<tr>
<td>Series.truncate</td>
<td>Function truncate a sorted DataFrame / Series before and/or after</td>
</tr>
</tbody>
</table>

**pandas.Series.align**

**Series.align** *(other=’outer’, level=None, copy=True, fill_value=None, method=None, limit=None)*

Align two Series object with the specified join method

- **Parameters**
  - `other`: Series
  - `join`: {‘outer’, ‘inner’, ‘left’, ‘right’}, default ‘outer’
  - `level`: int or name
    
    Broadcast across a level, matching Index values on the passed MultiIndex level
  - `copy`: boolean, default True
    
    Always return new objects. If copy=False and no reindexing is required, the same object will be returned (for better performance)
  - `fill_value`: object, default None
  - `method`: str, default ‘pad’
  - `limit`: int, default None
fill_value, method, inplace, limit are passed to fillna

Returns (left, right) : (Series, Series)
Aligned Series

pandas.Series.drop

Series.drop(labels, axis=0, level=None)
Return new object with labels in requested axis removed

Parameters
labels : array-like
axis : int
level : int or name, default None
For MultiIndex

Returns dropped : type of caller

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

Examples

ts.last(‘10D’) -> First 10 days

pandas.Series.head

Series.head(n=5)
Returns first n rows of Series

pandas.Series.idxmax

Series.idxmax(axis=0, out=None, skipna=True)
Index of first occurrence of maximum of values.

Parameters
skipna : boolean, default True
Exclude NA/null values

Returns idmax : Index of minimum of values
pandas.Series.idxmin

Series.\texttt{idxmin}(axis=None, out=None, skipna=True)
Index of first occurrence of minimum of values.

- **Parameters**: 
  - \texttt{skipna} : boolean, default True
    - Exclude NA/null values

- **Returns**: 
  - \texttt{idxmin} : Index of minimum of values

pandas.Series.isin

Series.\texttt{isin}(values)
Return boolean vector showing whether each element in the Series is exactly contained in the passed sequence of values

- **Parameters**: 
  - \texttt{values} : sequence

- **Returns**: 
  - \texttt{isin} : Series (boolean dtype)

pandas.Series.last

Series.\texttt{last}(offset)
Convenience method for subsetting final periods of time series data based on a date offset

- **Parameters**: 
  - \texttt{offset} : string, DateOffset, dateutil.relativedelta

- **Returns**: 
  - \texttt{subset} : type of caller

**Examples**

\texttt{ts.last(‘5M’)} -> Last 5 months

pandas.Series.reindex

Series.\texttt{reindex}(index=None, method=None, level=None, fill_value=nan, limit=None, copy=True)
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 \texttt{copy=False}

- **Parameters**: 
  - \texttt{index} : array-like or Index
    - New labels / index to conform to. Preferably an Index object to avoid duplicating data
    - 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
  - \texttt{copy} : boolean, default True
    - Return a new object, even if the passed indexes are the same
  - \texttt{level} : int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level
  - \texttt{fill_value} : scalar, default NaN
Value to use for missing values. Defaults to NaN, but can be any “compatible” value

**limit**: int, default None
Maximum size gap to forward or backward fill

**Returns** reindexed: Series

### pandas.Series.reindex_like

**Series.reindex_like**(other, method=None, limit=None, fill_value=nan)
Reindex Series to match index of another Series, optionally with filling logic

**Parameters**

- **other**: Series
- **method**: string or None
  See Series.reindex docstring
- **limit**: int, default None
  Maximum size gap to forward or backward fill

**Returns** reindexed: Series

**Notes**

Like calling s.reindex(other.index, method=...)

### pandas.Series.rename

**Series.rename**(mapper, inplace=False)
Alter Series index using dict or function

**Parameters**

- **mapper**: dict-like or function
  Transformation to apply to each index

**Returns** renamed: Series (new object)

**Notes**

Function / dict values must be unique (1-to-1)

**Examples**

```python
>>> x
foo 1
bar 2
baz 3

>>> x.rename(str.upper)
FOO 1
BAR 2
BAZ 3
```
>>> x.rename({"foo" : 'a', 'bar' : 'b', 'baz' : 'c'})
a 1
b 2
c 3

pandas.Series.reset_index

Series.
reset_index

Analogous to the DataFrame.reset_index function, see docstring there.

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
name : object, default None
    The name of the column corresponding to the Series values
inplace : boolean, default False
    Modify the Series in place (do not create a new object)

Returns

resetted : DataFrame, or Series if drop == True

pandas.Series.select

Series.
select

Return data corresponding to axis labels matching criteria

Parameters

crit : function
    To be called on each index (label). Should return True or False
axis : int

Returns

selection : type of caller

pandas.Series.take

Series.
take

Analogous to ndarray.take, return Series corresponding to requested indices

Parameters

indices : list / array of ints

Returns

taken : Series

pandas.Series.tail

Series.
tail

Returns last n rows of Series
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

### pandas.Series.truncate

Series` . `truncate` (before=None, after=None, copy=True)`

Function truncate a sorted DataFrame / Series before and/or after some particular dates.

- **Parameters**
  - `before` : date
    Truncate before date
  - `after` : date
    Truncate after date

- **Returns**
  - `truncated` : type of caller

### 21.2.8 Missing data handling

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.dropna()</code></td>
<td>Return Series without null values</td>
</tr>
<tr>
<td><code>Series.fillna(value=None, method=None, inplace=False, limit=None)</code></td>
<td>Fill NA/NaN values using the specified method</td>
</tr>
<tr>
<td><code>Series.interpolate(method)</code></td>
<td>Interpolate missing values (after the first valid value)</td>
</tr>
</tbody>
</table>

### pandas.Series.dropna

Series` . `dropna()`

Return Series without null values

- **Returns**
  - `valid` : Series

### pandas.Series.fillna

Series` . `fillna(value=None, method=None, inplace=False, limit=None)`

Fill NA/NaN values using the specified method

- **Parameters**
  - `value` : any kind (should be same type as array)
    Value to use to fill holes (e.g. 0)
  - `method` : {'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'
    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
  - `inplace` : boolean, default False
    If True, fill the Series in place. Note: this will modify any other views on this Series, for example a column in a DataFrame. Returns a reference to the filled object, which is self if inplace=True
  - `limit` : int, default None
    Maximum size gap to forward or backward fill

- **Returns**
  - `filled` : Series

See Also:

- `reindex`, `asfreq`
**pandas.Series.interpolate**

Series.interpolate(method='linear')
Interpolate missing values (after the first valid value)

**Parameters**
- `method`: {'linear', 'time', 'values'}
  Interpolation method. ‘time’ interpolation works on daily and higher resolution data to interpolate given length of interval ‘values’ using the actual index numeric values

**Returns**
- `interpolated`: Series

**21.2.9 Reshaping, sorting**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.argsort</td>
<td>Sorts values and index labels by value, in place.</td>
</tr>
<tr>
<td>Series.order</td>
<td>Sorts Series object, by value, maintaining index-value link</td>
</tr>
<tr>
<td>Series.reorder_levels</td>
<td>Rearranges index levels using input order.</td>
</tr>
<tr>
<td>Series.sort</td>
<td>Sort values and index labels by value, in place.</td>
</tr>
<tr>
<td>Series.sort_index</td>
<td>Sorts object by labels (along an axis)</td>
</tr>
<tr>
<td>Series.sortlevel</td>
<td>Sort Series with MultiIndex by chosen level. Data will be</td>
</tr>
<tr>
<td>Series.swaplevel</td>
<td>Swap levels i and j in a MultiIndex</td>
</tr>
<tr>
<td>Series.unstack</td>
<td>Unstack, a.k.a.</td>
</tr>
</tbody>
</table>

**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

**pandas.Series.order**

Series.order(na_last=True, ascending=True, kind='mergesort')
Sorts Series object, by value, maintaining index-value link

**Parameters**
- `na_last`: boolean (optional, default=True)
  Put NaN’s at beginning or end
- `ascending`: boolean, default True
  Sort ascending. Passing False sorts descending
- `kind`: {'mergesort', 'quicksort', 'heapsort'}, default 'mergesort'
  Choice of sorting algorithm. See np.sort for more information. ‘mergesort’ is the only stable algorithm
Returns y : Series

**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 not by key)

axis: where to reorder levels:

Returns type of caller (new object):

**pandas.Series.sort**

Series.sort(axis=0, kind='quicksort', order=None)
Sort values and index labels by value, in place. For compatibility with ndarray API. No return value

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

**pandas.Series.sort_index**

Series.sort_index(ascending=True)
Sort object by labels (along an axis)

Parameters ascending: boolean or list, default True
Sort ascending vs. descending. Specify list for multiple sort orders

Returns sorted_obj : Series

Examples

>>> result1 = s.sort_index(ascending=False)
>>> result2 = s.sort_index(ascending=[1, 0])

**pandas.Series.sortlevel**

Series.sortlevel(level=0, ascending=True)
Sort Series with MultiIndex by chosen level. Data will be lexicographically sorted by the chosen level followed by the other levels (in order)

Parameters level: int
ascending: bool, default True

Returns sorted: Series
**pandas.Series.swaplevel**

`Series.swaplevel(i, j, 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

**pandas.Series.unstack**

`Series.unstack(level=-1)`  
Unstack, a.k.a. pivot, Series with MultiIndex to produce DataFrame  

**Parameters**  
`level`: int, string, or list of these, default last level  
Level(s) to unstack, can pass level name  

**Returns**  
`unstacked`: DataFrame

**Examples**

```python  
>>> s  
one a  1.  
one b  2.  
two a  3.  
two b  4.  

>>> s.unstack(level=-1)  
a   b  
one  1.  2.  
two  3.  4.  

>>> s.unstack(level=0)  
one two  
a  1.  2.  
b  3.  4.  
```  

**21.2.10 Combining / joining / merging**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Series.append</td>
<td>Concatenate two or more Series. The indexes must not overlap</td>
</tr>
<tr>
<td>Series.replace</td>
<td>Replace arbitrary values in a Series</td>
</tr>
<tr>
<td>Series.update</td>
<td>Modify Series in place using non-NA values from passed</td>
</tr>
</tbody>
</table>

**pandas.Series.append**

`Series.append(to_append[, verify_integrity])`  
Concatenate two or more Series. The indexes must not overlap  

**Parameters**  
`to_append`: Series or list/tuple of Series  
`verify_integrity`: boolean, default False

---

21.2. Series
If True, raise Exception on creating index with duplicates

**Returns** appended : Series

### pandas.Series.replace

`Series.replace(to_replace, value=None, method='pad', inplace=False, limit=None)`

Replace arbitrary values in a Series

**Parameters**

- **to_replace**: list or dict
  - list of values to be replaced or dict of replacement values
- **value**: anything
  - if to_replace is a list then value is the replacement value
- **method**: {'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'
  - 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
- **inplace**: boolean, default False
  - If True, fill the Series in place. Note: this will modify any other views on this Series, for example a column in a DataFrame. Returns a reference to the filled object, which is self if inplace=True
- **limit**: int, default None
  - Maximum size gap to forward or backward fill

**Returns** replaced : Series

See Also:

fillna, reindex, asfreq

**Notes**

replace does not distinguish between NaN and None

### pandas.Series.update

`Series.update(other)`

Modify Series in place using non-NA values from passed Series. Aligns on index

**Parameters**

- **other**: Series

### 21.2.11 Time series-related

- **Series.asfreq**(freq[, method, how, normalize]) Convert all TimeSeries inside to specified frequency using DateOffset
- **Series.asof**(where) Return last good (non-NaN) value in TimeSeries if value is NaN for
- **Series.shift**([, periods, freq, copy]) Shift the index of the Series by desired number of periods with an
- **Series.first_valid_index**() Return label for first non-NA/null value
- **Series.last_valid_index**() Return label for last non-NA/null value
pandas.Series.asfreq

```python
Series.asfreq(freq, method=None, how=None, normalize=False)
```

Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters**
- `freq`: DateOffset object, or string
- `method`: {'backfill', 'bfill', 'pad', 'ffill', None}
- `how`: {'start', 'end'}, default end
- `normalize`: bool, default False

**Returns**
- converted : type of caller

pandas.Series.asof

```python
Series.asof(where)
```

Return last good (non-NaN) value in TimeSeries if value is NaN for requested date.

If there is no good value, NaN is returned.

**Parameters**
- `where`: date or array of dates

**Returns**
- value or NaN :

**Notes**

Dates are assumed to be sorted

pandas.Series.shift

```python
Series.shift(periods=1, freq=None, copy=True, **kwds)
```

Shift the index of the Series by desired number of periods with an optional time offset

**Parameters**
- `periods`: int
- `freq`: DateOffset, timedelta, or offset alias string, optional

**Returns**
- shifted : Series
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

pandas.Series.first_valid_index

Series.first_valid_index()
    Return label for first non-NA/null value

pandas.Series.last_valid_index

Series.last_valid_index()
    Return label for last non-NA/null value

pandas.Series.weekday

Series.weekday

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)
    Convenience method for frequency conversion and resampling of regular time-series data.

Parameters
    rule : the offset string or object representing target conversion
    how : string, method for down- or re-sampling, default to ‘mean’ for
downsampling
    axis : int, optional, default 0
    fill_method : string, fill_method for upsampling, default None
    closed : {'right', 'left'}, default None
        Which side of bin interval is closed
    label : {'right', 'left'}, default None
        Which bin edge label to label bucket with
    convention : {'start', 'end', 's', 'e'}
    kind: “period”/”timestamp” :
    loffset: timedelta :
        Adjust the resampled time labels

limit: int, default None :
    Maximum size gap to when reindexing with fill_method

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
pandas.Series.tz_convert

Series.tz_convert tz, copy=True
Convert TimeSeries to target time zone

Parameters  tz : string or pytz.timezone object
               copy : boolean, default True

Also make a copy of the underlying data

Returns converted : TimeSeries

pandas.Series.tz_localize

Series.tz_localize tz, copy=True
Localize tz-naive TimeSeries to target time zone

Parameters  tz : string or pytz.timezone object
               copy : boolean, default True

Also make a copy of the underlying data

Returns localized : TimeSeries

21.2.12 Plotting

Series.hist(by, ax, grid, xlabelsize, ...)  Draw histogram of the input series using matplotlib
Series.plot(series[, label, kind, ...])  Plot the input series with the index on the x-axis using matplotlib

pandas.Series.hist

Series.hist(by=None, ax=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, **kwds)
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

21.2. Series
yrot : float, default None
rotation of y axis labels

kwds : keywords
To be passed to the actual plotting function

Notes
See matplotlib documentation online for more on this

pandas.Series.plot

Series.plot (series, label=None, kind='line', use_index=True, rot=None, xticks=None, yticks=None, xlim=None, ylim=None, ax=None, style=None, grid=None, legend=False, logx=False, logy=False, secondary_y=False, **kwds)
Plot the input series with the index on the x-axis using matplotlib

Parameters label : label argument to provide to plot
kind : {'line', 'bar', 'barh', 'kde', 'density'}
bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density Estimation plot
use_index : boolean, default True
Plot index as axis tick labels
rot : int, default None
Rotation for tick labels
xticks : sequence
Values to use for the xticks
yticks : sequence
Values to use for the yticks
xlim : 2-tuple/list
ylim : 2-tuple/list
ax : matplotlib axis object
If not passed, uses gca()
style : string, default matplotlib default
matplotlib line style to use
grid : matplotlib grid
legend: matplotlib legend : 
logx : boolean, default False
For line plots, use log scaling on x axis
logy : boolean, default False
For line plots, use log scaling on y axis
secondary_y : boolean or sequence of ints, default False
If True then y-axis will be on the right
kwds : keywords
Options to pass to matplotlib plotting method

Notes
See matplotlib documentation online for more on this subject

21.2.13 Serialization / IO / Conversion

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Series.from_csv(path[, sep, parse_dates, ...])</code></td>
<td>Read delimited file into Series</td>
</tr>
<tr>
<td><code>Series.load(path)</code></td>
<td></td>
</tr>
<tr>
<td><code>Series.save(path)</code></td>
<td></td>
</tr>
<tr>
<td><code>Series.to_csv(path[, index, sep, na_rep, ...])</code></td>
<td>Write Series to a comma-separated values (csv) file</td>
</tr>
<tr>
<td><code>Series.to_dict()</code></td>
<td>Convert Series to <code>{label -&gt; value}</code> dict</td>
</tr>
<tr>
<td><code>Series.to_sparse([kind, fill_value])</code></td>
<td>Convert Series to SparseSeries</td>
</tr>
<tr>
<td><code>Series.to_string([buf, na_rep, ...])</code></td>
<td>Render a string representation of the Series</td>
</tr>
</tbody>
</table>

**pandas.Series.from_csv**

```python
classmethod Series.from_csv(path, sep=' ', parse_dates=True, header=None, index_col=0, encoding=None)
```

Read delimited file into Series

**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 0
    - Row to use at 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

**Returns**
- `y` : Series

**pandas.Series.load**

```python
classmethod Series.load(path)
```

21.2. Series
pandas.Series.save

Series.save(path)

pandas.Series.to_csv

Series.to_csv(path, index=True, sep=’,’, na_rep=’’, float_format=None, header=False, index_label=None, mode=’w’, nanRep=None, encoding=None)
Write Series to a comma-separated values (csv) file

Parameters

path : string file path or file handle / StringIO
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

pandas.Series.to_dict

Series.to_dict()
Convert Series to {label -> value} dict

Returns value_dict : dict

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_string

Series.to_string(buf=None, na_rep='NaN', float_format=None, nanRep=None, length=False, dtype=False, name=False)

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

length : boolean, default False
Add the Series length

dtype : boolean, default False
Add the Series dtype

name : boolean, default False
Add the Series name (which may be None)

Returns
formatted : string (if not buffer passed)

21.3 DataFrame

21.3.1 Attributes and underlying data

Axes

- index: row labels
- columns: column labels

DataFrame.as_matrix([columns])  Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless a specific list of columns is provided.

DataFrame.dtypes

DataFrame.get_dtype_counts()  return the counts of dtypes in this frame

DataFrame.values  Convert the frame to its Numpy-array matrix representation. Columns

DataFrame.axes

DataFrame.ndim

DataFrame.shape

pandas.DataFrame.as_matrix

DataFrame.as_matrix(columns=None)

Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless a specific list of columns is provided.

NOTE: 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 accomodates all will be chosen use this with care if you are not dealing with the blocks
### pandas.DataFrame.dtypes

DataFrame.dtypes

`DataFrame.get_dtype_counts()` returns the counts of dtypes in this frame.

### pandas.DataFrame.values

DataFrame.values

Convert the frame to its Numpy-array matrix representation. Columns are presented in sorted order unless a specific list of columns is provided.

**NOTE:** 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 accomodates all will be chosen use this with care if you are not dealing with the blocks.

**e.g. if the dtypes are float16, float32 -> float32**  float16, float32, float64 -> float64 int32, uint8 -> int32

**Parameters** columns : array-like

Specific column order

**Returns** values : ndarray

If the DataFrame is heterogeneous and contains booleans or objects, the result will be of dtype=object

### pandas.DataFrame.axes

DataFrame.axes

### pandas.DataFrame.ndim

DataFrame.ndim

### pandas.DataFrame.shape

DataFrame.shape
### 21.3.2 Conversion / Constructors

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DataFrame.__init__</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>DataFrame.astype</code></td>
<td>Cast object to input numpy.dtype</td>
</tr>
<tr>
<td><code>DataFrame.convert_objects</code></td>
<td>Attempt to infer better dtype for object columns</td>
</tr>
<tr>
<td><code>DataFrame.copy</code></td>
<td>Make a copy of this object</td>
</tr>
</tbody>
</table>

#### pandas.DataFrame.__init__

`DataFrame.__init__(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

- **index**: Index or array-like
  
  Index to use for resulting frame. Will default to np.arange(n) if no indexing information part of input data and no index provided

- **columns**: Index or array-like
  
  Will default to np.arange(n) if not column labels provided

- **dtype**: dtype, default None
  
  Data type to force, otherwise 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_csv` from CSV files
- `DataFrame.from_items` from sequence of (key, value) pairs

**Examples**

```python
>>> d = {'col1': ts1, 'col2': ts2}
>>> df = DataFrame(data=d, index=index)
>>> df2 = DataFrame(np.random.randn(10, 5))
>>> df3 = DataFrame(np.random.randn(10, 5),
...                   columns=['a', 'b', 'c', 'd', 'e'])
```
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

pandas.DataFrame.astype

```
DataFrame.astype (dtype, copy=True, raise_on_error=True)
```

Cast object to input numpy.dtype Return a copy when copy = True (be really careful with this!)

**Parameters**
- `dtype`: numpy.dtype or Python type
- `raise_on_error`: raise on invalid input

**Returns**
- `casted`: type of caller

pandas.DataFrame.convert_objects

```
DataFrame.convert_objects (convert_dates=True, convert_numeric=False)
```

Attempt to infer better dtype for object columns Always returns a copy (even if no object columns)

**Parameters**
- `convert_dates`: if True, attempt to soft convert_dates, if ‘coerce’, force conversion (and non-convertibles get NaT)
- `convert_numeric`: if True attempt to coerce to numerbers (including strings), non-convertibles get NaN

**Returns**
- `converted`: DataFrame

pandas.DataFrame.copy

```
DataFrame.copy (deep=True)
```

Make a copy of this object

**Parameters**
- `deep`: boolean, default True

Make a deep copy, i.e. also copy data

**Returns**
- `copy`: type of caller

21.3.3 Indexing, iteration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.head</td>
<td>Returns first n rows of DataFrame</td>
</tr>
<tr>
<td>DataFrame.ix</td>
<td>Insert column into DataFrame at specified location. Raises Exception if</td>
</tr>
<tr>
<td>DataFrame.insert</td>
<td>Insert column into DataFrame at specified location. Raises Exception if</td>
</tr>
<tr>
<td>DataFrame.<strong>iter</strong></td>
<td>Iterate over columns of the frame.</td>
</tr>
<tr>
<td>DataFrame.iteritems()</td>
<td>Iterate over (column, series) pairs</td>
</tr>
<tr>
<td>DataFrame.iterrows()</td>
<td>Iterate over rows of DataFrame as (index, Series) pairs</td>
</tr>
<tr>
<td>DataFrame.itertuples((index))</td>
<td>Iterate over rows of DataFrame as tuples, with index value</td>
</tr>
<tr>
<td>DataFrame.lookup</td>
<td>Label-based “fancy indexing” function for DataFrame. Given equal-length</td>
</tr>
<tr>
<td>DataFrame.pop</td>
<td>Return column and drop from frame.</td>
</tr>
<tr>
<td>DataFrame.tail</td>
<td>Returns last n rows of DataFrame</td>
</tr>
<tr>
<td>DataFrame.xs</td>
<td>Returns a cross-section (row(s) or column(s)) from the DataFrame.</td>
</tr>
</tbody>
</table>

pandas.DataFrame.head

```
DataFrame.head (n=5)
```

Returns first n rows of DataFrame
pandas.DataFrame.ix

DataFrame.ix

pandas.DataFrame.insert

DataFrame.insert(loc, column, value)
Insert column into DataFrame at specified location. Raises Exception if column is already contained in the DataFrame

Parameters:
- loc : int
  Must have 0 <= loc <= len(columns)
- column : object
- value : int, Series, or array-like

pandas.DataFrame.__iter__

DataFrame.__iter__()
Iterate over columns of the frame.

pandas.DataFrame.iteritems

DataFrame.iteritems()
Iterator over (column, series) pairs

pandas.DataFrame.itertuples

DataFrame.itertuples(index=True)
Iterate over rows of DataFrame as tuples, with index value as first element of the tuple

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
- col_labels : sequence
Notes

Akin to

result = [] for row, col in zip(row_labels, col_labels):
    result.append(df.get_value(row, col))

**pandas.DataFrame.pop**

DataFrame.pop(*item*)

Return column and drop from frame. Raise KeyError if not found.

Returns column : Series

**pandas.DataFrame.tail**

DataFrame.tail(*n=5*)

Returns last n rows of DataFrame

**pandas.DataFrame.xs**

DataFrame.xs(*key*, **axis=0, level=None, copy=True*)

Returns a cross-section (row(s) or column(s)) from the 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.

copy : boolean, default True

Whether to make a copy of the data

Returns xs : Series or DataFrame

Examples

```
>>> df
   A  B  C
a  4  5  2
b  4  0  9
c  9  7  3
>>> df.xs('a')
   A  B  C
Name: a
```
21.3.4 Binary operator functions

```
DataFrame.add(other[, axis, level, fill_value]) Binary operator add with support to substitute a fill_value for missing data in one of the inputs

DataFrame.div(other[, axis, level, fill_value]) Binary operator divide with support to substitute a fill_value for missing data in one of the inputs

DataFrame.mul(other[, axis, level, fill_value]) Binary operator multiply with support to substitute a fill_value for missing data in one of the inputs

DataFrame.sub(other[, axis, level, fill_value]) Binary operator subtract with support to substitute a fill_value for missing data in one of the inputs

DataFrame.radd(other[, axis, level, fill_value]) Binary operator radd with support to substitute a fill_value for missing data in one of the inputs

DataFrame.rdiv(other[, axis, level, fill_value]) Binary operator rdivide with support to substitute a fill_value for missing data in one of the inputs

DataFrame.rmul(other[, axis, level, fill_value]) Binary operator rmultiply with support to substitute a fill_value for missing data in one of the inputs

DataFrame.rsub(other[, axis, level, fill_value]) Binary operator rsubtract with support to substitute a fill_value for missing data in one of the inputs

DataFrame.combine(other, func[, fill_value, ...]) Add two DataFrame objects and do not propagate NaN values, so if for a

DataFrame.combineAdd(other) Add two DataFrame objects and do not propagate

DataFrame.combineFirst(other) Combine two DataFrame objects and default to non-null values in frame

DataFrame.combineMult(other) Multiply two DataFrame objects and do not propagate NaN values, so if
```

pandas.DataFrame.add

```
DataFrame.add (other, axis='columns', level=None, fill_value=None)
Binary operator add 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  
  
  `fill_value` : None or float value, default None  
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing  
  
  `level` : int or name  
  Broadcast across a level, matching Index values on the passed MultiIndex level  
  
Returns  `result` : DataFrame  

Notes  
Mismatched indices will be unioned together  

pandas.DataFrame.div  

DataFrame.div( `other` , axis=’columns’ , level=None , fill_value=None )  
Binary operator divide 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  
  
  `fill_value` : None or float value, default None  
  Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing  
  
  `level` : int or name  
  Broadcast across a level, matching Index values on the passed MultiIndex level  
  
Returns  `result` : DataFrame  

Notes  
Mismatched indices will be unioned together  

pandas.DataFrame.mul  

DataFrame.mul( `other` , axis=’columns’ , level=None , fill_value=None )  
Binary operator multiply 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  
  
  `fill_value` : None or float value, default None
Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing.

**level**: int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**  **result**: DataFrame

**Notes**

Mismatched indices will be unioned together.

### pandas.DataFrame.sub

DataFrame.sub(\(other, axis='columns', level=None, fill_value=None\))

Binary operator subtract 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.

**fill_value**: None or float value, default None.

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing.

**level**: int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**  **result**: DataFrame

**Notes**

Mismatched indices will be unioned together.

### pandas.DataFrame.radd

DataFrame.radd(\(other, axis='columns', level=None, fill_value=None\))

Binary operator radd 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.

**fill_value**: None or float value, default None.

Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing.

**level**: int or name

Broadcast across a level, matching Index values on the passed MultiIndex level.

**Returns**  **result**: DataFrame
Notes

Mismatched indices will be unioned together

**pandas.DataFrame.rdiv**

DataFrame.

\[
\text{rdiv}(\text{other, axis}='\text{columns}', \text{level}=\text{None, fill_value}=\text{None})
\]

Binary operator rdivide 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
  - **fill_value**: None or float value, default None
    - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
  - **level**: int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

result : DataFrame

Notes

Mismatched indices will be unioned together

**pandas.DataFrame.rmul**

DataFrame.

\[
\text{rmul}(\text{other, axis}='\text{columns}', \text{level}=\text{None, fill_value}=\text{None})
\]

Binary operator rmultiply 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
  - **fill_value**: None or float value, default None
    - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
  - **level**: int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**

result : DataFrame

Notes

Mismatched indices will be unioned together
pandas.DataFrame.rsub

```
DataFrame.rsub other, axis='columns', level=None, fill_value=None
```

Binary operator rsubtract 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
  - **fill_value**: None or float value, default None
    - Fill missing (NaN) values with this value. If both DataFrame locations are missing, the result will be missing
  - **level**: int or name
    - Broadcast across a level, matching Index values on the passed MultiIndex level

**Returns**
- **result**: DataFrame

**Notes**
Mismached indices will be unioned together

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
  - **fill_value**: scalar value
  - **overwrite**: boolean, default True
    - If True then overwrite values for common keys in the calling frame

**Returns**
- **result**: DataFrame

pandas.DataFrame.combineAdd

```
DataFrame.combineAdd other
```

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

**Returns**
- **DataFrame**

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
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

Parameters `other` : DataFrame  
Returns `combined` : DataFrame

Examples

```python
>>> a.combine_first(b)
a’s values prioritized, use values from b to fill holes
```

**pandas.DataFrame.combineMult**

DataFrame `combineMult(other)`
Multiply 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
Returns `DataFrame`

**21.3.5 Function application, GroupBy**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.apply</td>
<td>Applies function along input axis of DataFrame. Objects passed to functions are Series objects having index either the DataFrame’s index (axis=0) or the columns (axis=1). Return type depends on whether passed function aggregates values.</td>
</tr>
<tr>
<td>DataFrame.applymap</td>
<td>Apply a function to a DataFrame that is intended to operate elementwise.</td>
</tr>
<tr>
<td>DataFrame.groupby</td>
<td>Group series using mapper (dict or key function, apply given function to each group)</td>
</tr>
</tbody>
</table>

**pandas.DataFrame.apply**

DataFrame `apply(func[, axis, broadcast, ...])`  
Applies function along input axis of DataFrame. Objects passed to functions are Series objects having index either the DataFrame’s index (axis=0) or the columns (axis=1). Return type depends on whether passed function aggregates values.

Parameters `func` : function
Function to apply to each column

axis : {0, 1}
0 : apply function to each column 1 : apply function to each row

broadcast : bool, default False
For aggregation functions, return object of same size with values propagated

raw : boolean, default False
If False, convert each row or column into a Series. If raw=True the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance

args : tuple
Positional arguments to pass to function in addition to the array/series

Additional keyword arguments will be passed as keywords to the function

Returns `applied` : Series or DataFrame
See Also:

**DataFrame.applymap** For elementwise operations

Examples

```python
df.apply(numpy.sqrt)  # returns DataFrame
>>> df.apply(numpy.sum, axis=0)  # equiv to df.sum(0)
>>> df.apply(numpy.sum, axis=1)  # equiv to df.sum(1)
```

**pandas.DataFrame.applymap**

Dataframe.applymap(func)

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing map(func, series) for each series in the DataFrame

Parameters

- **func**: function
  
  Python function, returns a single value from a single value

Returns

- **applied**: DataFrame

**pandas.DataFrame.groupby**

Dataframe.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=True)

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 / list of functions, dict, Series, or tuple / list of column names. Called on each element of the object index to determine the groups. If a dict or Series is passed, the Series or dict VALUES will be used to determine the groups
- **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
- **group_keys**: boolean, default True
  
  When calling apply, add group keys to index to identify pieces

Returns

- **GroupBy object**
Examples

# DataFrame result >>> data.groupby(func, axis=0).mean()
# DataFrame result >>> data.groupby(['col1', 'col2'])['col3'].mean()
# DataFrame with hierarchical index >>> data.groupby(['col1', 'col2']).mean()

21.3.6 Computations / Descriptive Stats

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pandas.DataFrame.abs()</td>
<td>Return an object with absolute value taken.</td>
</tr>
<tr>
<td>pandas.DataFrame.any(</td>
<td>Return whether any element is True over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.clip(</td>
<td>Trim values at input threshold(s)</td>
</tr>
<tr>
<td>pandas.DataFrame.clip_lower(</td>
<td>Trim values below threshold</td>
</tr>
<tr>
<td>pandas.DataFrame.clip_upper(</td>
<td>Trim values above threshold</td>
</tr>
<tr>
<td>pandas.DataFrame.corr</td>
<td>Compute pairwise correlation of columns, excluding NA/null values</td>
</tr>
<tr>
<td>pandas.DataFrame.corrwith</td>
<td>Compute pairwise correlation between rows or columns of two DataFrame</td>
</tr>
<tr>
<td>pandas.DataFrame.count</td>
<td>Return Series with number of non-NA/null observations over requested axis</td>
</tr>
<tr>
<td>pandas.DataFrame.cov(</td>
<td>Compute pairwise covariance of columns, excluding NA/null values</td>
</tr>
<tr>
<td>pandas.DataFrame.cummax</td>
<td>Return DataFrame of cumulative max over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.cummin</td>
<td>Return DataFrame of cumulative min over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.cumprod</td>
<td>Return cumulative product over requested axis as DataFrame</td>
</tr>
<tr>
<td>pandas.DataFrame.cumsum</td>
<td>Return DataFrame of cumulative sums over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.describe</td>
<td>Generate various summary statistics of each column, excluding</td>
</tr>
<tr>
<td>pandas.DataFrame.diff(</td>
<td>1st discrete difference of object</td>
</tr>
<tr>
<td>pandas.DataFrame.kurt</td>
<td>Return unbiased kurtosis over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.max</td>
<td>Return maximum over requested axis</td>
</tr>
<tr>
<td>pandas.DataFrame.mean</td>
<td>Return mean over requested axis</td>
</tr>
<tr>
<td>pandas.DataFrame.median</td>
<td>Return median over requested axis</td>
</tr>
<tr>
<td>pandas.DataFrame.min</td>
<td>Return minimum over requested axis</td>
</tr>
<tr>
<td>pandas.DataFrame.pct_change(</td>
<td>Percent change over given number of periods</td>
</tr>
<tr>
<td>pandas.DataFrame.prod(</td>
<td>Return product over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.quantile</td>
<td>Return values at the given quantile over requested axis, a la</td>
</tr>
<tr>
<td>pandas.DataFrame.rank(</td>
<td>Compute numerical data ranks (1 through n) along axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.skew</td>
<td>Return unbiased skewness over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.sum(</td>
<td>Return sum over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.std(</td>
<td>Return standard deviation over requested axis.</td>
</tr>
<tr>
<td>pandas.DataFrame.var</td>
<td>Return variance over requested axis.</td>
</tr>
</tbody>
</table>

pandas.DataFrame.abs

DataFrame.abs()

Return an object with absolute value taken. Only applicable to objects that are all numeric.

Returns abs: type of caller

pandas.DataFrame.any

DataFrame.any(axis=0, bool_only=None, skipna=True, level=None)

Return whether any element is True over requested axis. % (na_action)
**Parameters**  
axis : {0, 1}  
0 for row-wise, 1 for column-wise  
skipna : boolean, default True  
Exclude NA/null values. If an entire row/column is NA, the result will be NA  
level : int, default None  
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame  
bool_only : boolean, default None  
Only include boolean data.  

**Returns**  
any : Series (or DataFrame if level specified)

---

### pandas.DataFrame.clip

DataFrame.clip(lower=None, upper=None)  
Trim values at input threshold(s)  

**Parameters**  
lower : float, default None  
upper : float, default None  

**Returns**  
clipped : DataFrame

---

### pandas.DataFrame.clip_lower

DataFrame.clip_lower(threshold)  
Trim values below threshold  

**Returns**  
clipped : DataFrame

---

### pandas.DataFrame.clip_upper

DataFrame.clip_upper(threshold)  
Trim values above threshold  

**Returns**  
clipped : DataFrame

---

### pandas.DataFrame.corr

DataFrame.corr(method='pearson', min_periods=None)  
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 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
  - **axis**: {0, 1}
    - 0 to compute column-wise, 1 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)
Return Series with number of non-NA/null observations over requested axis. Works with non-floating point data as well (detects NaN and None)

**Parameters**
- **axis**: {0, 1}
  - 0 for row-wise, 1 for column-wise
- **level**: int, default None
  - If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame
- **numeric_only**: boolean, default False
  - Include only float, int, boolean data

**Returns**
- **count**: Series (or DataFrame if level specified)

pandas.DataFrame.cov

DataFrame.cov(min_periods=None)
Compute pairwise covariance of columns, excluding NA/null values

**Parameters**
- **min_periods**: int, optional
  - Minimum number of observations required per pair of columns to have a valid result.

**Returns**
- **y**: DataFrame
  - y contains the covariance matrix of the DataFrame’s time series.
    - The covariance is normalized by N-1 (unbiased estimator).

pandas.DataFrame.cummax

DataFrame.cummax(axis=None, skipna=True)
Return DataFrame of cumulative max over requested axis.

**Parameters**
- **axis**: {0, 1}
  - 0 for row-wise, 1 for column-wise
skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns y : DataFrame

pandas.DataFrame.cummin

DataFrame.cummin(axis=None, skipna=True)

Return DataFrame of cumulative min over requested axis.

Parameters axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns y : DataFrame

pandas.DataFrame.cumprod

DataFrame.cumprod(axis=None, skipna=True)

Return cumulative product over requested axis as DataFrame

Parameters axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns y : DataFrame

pandas.DataFrame.cumsum

DataFrame.cumsum(axis=None, skipna=True)

Return DataFrame of cumulative sums over requested axis.

Parameters axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

Returns y : DataFrame

pandas.DataFrame.describe

DataFrame.describe(percentile_width=50)

Generate various summary statistics of each column, excluding NaN values. These include: count, mean, std, 
min, max, and lower%/50%/upper% percentiles

Parameters percentile_width : float, optional

width of the desired uncertainty interval, default is 50, which corresponds to lower=25, upper=75

21.3. DataFrame
Returns DataFrame of summary statistics:

pandas.DataFrame.diff

DataFrame.\texttt{diff}(periods=1)
1st discrete difference of object

\textbf{Parameters} \texttt{periods} : int, default 1
Periods to shift for forming difference

\textbf{Returns} \texttt{diffed} : DataFrame

pandas.DataFrame.kurt

DataFrame.\texttt{kurt}(axis=0, skipna=True, level=None)
Return unbiased kurtosis over requested axis. NA/null values are excluded

\textbf{Parameters} \texttt{axis} : \{0, 1\}
0 for row-wise, 1 for column-wise

\texttt{skipna} : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

\texttt{level} : int, default None
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

\textbf{Returns} \texttt{kurt} : Series (or DataFrame if level specified)

pandas.DataFrame.mad

DataFrame.\texttt{mad}(axis=0, skipna=True, level=None)
Return mean absolute deviation over requested axis. NA/null values are excluded

\textbf{Parameters} \texttt{axis} : \{0, 1\}
0 for row-wise, 1 for column-wise

\texttt{skipna} : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

\texttt{level} : int, default None
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

\textbf{Returns} \texttt{mad} : Series (or DataFrame if level specified)

pandas.DataFrame.max

DataFrame.\texttt{max}(axis=0, skipna=True, level=None)
Return maximum over requested axis. NA/null values are excluded

\textbf{Parameters} \texttt{axis} : \{0, 1\}
0 for row-wise, 1 for column-wise
skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

Returns max : Series (or DataFrame if level specified)

pandas.DataFrame.mean

DataFrame.mean (axis=0, skipna=True, level=None)

Return mean over requested axis. NA/null values are excluded

Parameters axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

Returns mean : Series (or DataFrame if level specified)

pandas.DataFrame.median

DataFrame.median (axis=0, skipna=True, level=None)

Return median over requested axis. NA/null values are excluded

Parameters axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

Returns median : Series (or DataFrame if level specified)

pandas.DataFrame.min

DataFrame.min (axis=0, skipna=True, level=None)

Return minimum over requested axis. NA/null values are excluded

Parameters axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

Returns min : Series (or DataFrame if level specified)
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

```
level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a DataFrame

Returns  min : Series (or DataFrame if level specified)

pandas.DataFrame.pct_change

DataFrame.pct_change(periods=1, fill_method='pad', limit=None, freq=None, **kwds)

Percent change over given number of periods

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())

Returns  chg : Series or DataFrame

pandas.DataFrame.prod

DataFrame.prod(axis=0, skipna=True, level=None)

Return product over requested axis. NA/null values are treated as 1

Parameters  axis : {0, 1}

0 for row-wise, 1 for column-wise

skipna : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

level : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a DataFrame

Returns  product : Series (or DataFrame if level specified)

pandas.DataFrame.quantile

DataFrame.quantile(q=0.5, axis=0, numeric_only=True)

Return values at the given quantile over requested axis, a la scoreatpercentile in scipy.stats

Parameters  q : quantile, default 0.5 (50% quantile)

0 <= q <= 1

axis : {0, 1}

0 for row-wise, 1 for column-wise
```
Returns quantiles : Series

pandas.DataFrame.rank

DataFrame.rank (axis=0, numeric_only=None, method='average', na_option='keep', ascending=True)
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, 1}, default 0
Ranks over columns (0) or rows (1)
numeric_only : boolean, default None
Include only float, int, boolean data
method : {'average', 'min', 'max', 'first'}
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
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)

Returns ranks : DataFrame

pandas.DataFrame.skew

DataFrame.skew (axis=0, skipna=True, level=None)
Return unbiased skewness over requested axis. NA/null values are excluded

Parameters axis : {0, 1}
0 for row-wise, 1 for column-wise
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA
level : int, default None
If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into
a DataFrame

Returns skew : Series (or DataFrame if level specified)

pandas.DataFrame.sum

DataFrame.sum (axis=0, numeric_only=None, skipna=True, level=None)
Return sum over requested axis. NA/null values are excluded

Parameters axis : {0, 1}
0 for row-wise, 1 for column-wise
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int, 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 data. If None, will attempt to use everything, then use only numeric data

**Returns** sum : Series (or DataFrame if level specified)

**pandas.DataFrame.std**

DataFrame.std(axis=0, skipna=True, level=None, ddof=1)

Return standard deviation over requested axis. NA/null values are excluded

**Parameters**

**axis** : {0, 1}

0 for row-wise, 1 for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** std : Series (or DataFrame if level specified)

Normalized by N-1 (unbiased estimator).

**pandas.DataFrame.var**

DataFrame.var(axis=0, skipna=True, level=None, ddof=1)

Return variance over requested axis. NA/null values are excluded

**Parameters**

**axis** : {0, 1}

0 for row-wise, 1 for column-wise

**skipna** : boolean, default True

Exclude NA/null values. If an entire row/column is NA, the result will be NA

**level** : int, default None

If the axis is a MultiIndex (hierarchical), count along a particular level, collapsing into a DataFrame

**Returns** var : Series (or DataFrame if level specified)

Normalized by N-1 (unbiased estimator).
21.3.7 Reindexing / Selection / Label manipulation

- **DataFrame.add_prefix**<sup>(prefix)</sup> Concatenate prefix string with panel items names.
- **DataFrame.add_suffix**<sup>(suffix)</sup> Concatenate suffix string with panel items names.
- **DataFrame.align**<sup>(other[, join, axis, level, ...])</sup> Align two DataFrame object on their index and columns with the specified join method for each axis Index.
- **DataFrame.drop**<sup>(labels[, axis, level])</sup> Return new object with labels in requested axis removed.
- **DataFrame.drop_duplicates**<sup>([cols, take_last, ...])</sup> Return DataFrame with duplicate rows removed, optionally only.
- **DataFrame.duplicated**<sup>([cols, take_last])</sup> Return boolean Series denoting duplicate rows, optionally only.
- **DataFrame.filter**<sup>([items, like, regex])</sup> Restrict frame’s columns to set of items or wildcard.
- **DataFrame.first**<sup>(offset)</sup> Convenience method for subsetting initial periods of time series data.
- **DataFrame.head**<sup>([n])</sup> Returns first n rows of DataFrame.
- **DataFrame.idxmax**<sup>([axis, skipna])</sup> Return index of first occurrence of maximum over requested axis.
- **DataFrame.idxmin**<sup>([axis, skipna])</sup> Return index of first occurrence of minimum over requested axis.
- **DataFrame.last**<sup>(offset)</sup> Convenience method for subsetting final periods of time series data.
- **DataFrame.reindex**<sup>([index, columns, method, ...])</sup> Conform DataFrame to new index with optional filling logic, placing.
- **DataFrame.reindex_axis**<sup>(labels[, axis, ...])</sup> Conform DataFrame to new index with optional filling logic, placing.
- **DataFrame.reindex_like**<sup>(other[, method, ...])</sup> Reindex DataFrame to match indices of another DataFrame, optionally.
- **DataFrame.rename**<sup>([index, columns, copy, inplace])</sup> Alter index and / or columns using input function or functions.
- **DataFrame.reset_index**<sup>([level, drop, ...])</sup> For DataFrame with multi-level index, return new DataFrame with.
- **DataFrame.select**<sup>(crit[, axis])</sup> Return data corresponding to axis labels matching criteria.
- **DataFrame.set_index**<sup>(keys[, drop, append, ...])</sup> Set the DataFrame index (row labels) using one or more existing.
- **DataFrame.tail**<sup>([n])</sup> Returns last n rows of DataFrame.
- **DataFrame.take**<sup>(indices[, axis])</sup> Analogous to ndarray.take, return DataFrame corresponding to requested.
- **DataFrame.truncate**<sup>([before, after, copy])</sup> Function truncate a sorted DataFrame / Series before and/or after.

### pandas.DataFrame.add_prefix

**DataFrame.add_prefix**<sup>(prefix)</sup>

Concatenate prefix string with panel items names.

**Parameters**

- `prefix` : string

**Returns**

- `with_prefix` : type of caller

### pandas.DataFrame.add_suffix

**DataFrame.add_suffix**<sup>(suffix)</sup>

Concatenate suffix string with panel items names.

**Parameters**

- `suffix` : string

**Returns**

- `with_suffix` : type of caller

### pandas.DataFrame.align

**DataFrame.align**<sup>(other, join='outer', axis=None, level=None, copy=True, fill_value=None, method=None, limit=None, fill_axis=0)</sup>

Align two DataFrame object on their index and columns with the specified join method for each axis Index.
Parameters  other : DataFrame or Series
  join : {'outer', 'inner', 'left', 'right'}, default 'outer'
  axis : {0, 1, None}, default None
    Align on index (0), columns (1), or both (None)
  level : int or name
    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, 1}, default 0
    Filling axis, method and limit

Returns  (left, right) : (DataFrame, type of other)
  Aligned objects

pandas.DataFrame.drop

DataFrame.drop (labels, axis=0, level=None)
  Return new object with labels in requested axis removed

Parameters  labels : array-like
  axis : int
  level : int or name, default None
    For MultiIndex

Returns  dropped : type of caller

pandas.DataFrame.drop_duplicates

DataFrame.drop_duplicates (cols=None, take_last=False, inplace=False)
  Return DataFrame with duplicate rows removed, optionally only considering certain columns

Parameters  cols : column label or sequence of labels, optional
  Only consider certain columns for identifying duplicates, by default use all of the
  columns
  take_last : boolean, default False
    Take the last observed row in a row. Defaults to the first row
  inplace : boolean, default False
    Whether to drop duplicates in place or to return a copy
**pandas.DataFrame.duplicated**

`DataFrame.duplicated(cols=None, take_last=False)`  
Return boolean Series denoting duplicate rows, optionally only considering certain columns  

**Parameters**  
- `cols` : column label or sequence of labels, optional  
  - Only consider certain columns for identifying duplicates, by default use all of the columns  
- `take_last` : boolean, default False  
  - Take the last observed row in a row. Defaults to the first row

**Returns**  
- `duplicated` : Series

**pandas.DataFrame.filter**

`DataFrame.filter(items=None, like=None, regex=None)`  
Restrict frame’s columns to set of items or wildcard  

**Parameters**  
- `items` : list-like  
  - List of columns to restrict to (must not all be present)  
- `like` : string  
  - Keep columns where “arg in col == True”  
- `regex` : string (regular expression)  
  - Keep columns with re.search(regex, col) == True

**Returns**  
- `DataFrame with filtered columns`

**Notes**

Arguments are mutually exclusive, but this is not checked for

**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

**Examples**

ts.last(‘10D’) -> First 10 days
pandas.DataFrame.head

DataFrame.head(n=5)

Returns first n rows of DataFrame

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, 1}
0 for row-wise, 1 for column-wise
skipna : boolean, default True
Exclude NA/null values. If an entire row/column is NA, the result will be first index.

Returns  idxmax : Series

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, 1}
0 for row-wise, 1 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

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

Examples

ts.last(‘5M’) -> Last 5 months

pandas.DataFrame.reindex

DataFrame.reindex(index=None, columns=None, method=None, level=None, fill_value=nan, limit=None, copy=True)

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  
index : array-like, optional
New labels / index to conform to. Preferably an Index object to avoid duplicating data

**columns**: array-like, optional

Same usage as index argument

**method**: \{'backfill', 'bfill', 'pad', 'ffill', None\}, default None

Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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 size gap to forward or backward fill

**Returns**

reindexed : same type as calling instance

**Examples**

```python
>>> df.reindex(index=[date1, date2, date3], columns=['A', 'B', 'C'])
```

**pandas.DataFrame.reindex_axis**

DataFrame.reindex_axis(labels, axis=0, method=None, level=None, copy=True, limit=None, fill_value=nan)

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**

**index**: array-like, optional

New labels / index to conform to. Preferably an Index object to avoid duplicating data

**axis**: \{0, 1\}

0 \(\rightarrow\) index (rows) 1 \(\rightarrow\) columns

**method**: \{'backfill', 'bfill', 'pad', 'ffill', None\}, default None

Method to use for filling holes in reindexed DataFrame pad / ffill: propagate last valid observation forward to next valid backfill / bfill: use NEXT valid observation 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 size gap to forward or backward fill
Returns reindexed : same type as calling instance

See Also:
DataFrame.reindex, DataFrame.reindex_like

Examples

```python
>>> df.reindex_axis(['A', 'B', 'C'], axis=1)
```

pandas.DataFrame.reindex_like

DataFrame.reindex_like(other, method=None, copy=True, limit=None, fill_value=nan)

Reindex DataFrame to match indices of another DataFrame, optionally with filling logic

Parameters
- other : DataFrame
- method : string or None
- copy : boolean, default True
- limit : int, default None
  Maximum size gap to forward or backward fill

Returns reindexed : DataFrame

Notes

Like calling s.reindex(index=other.index, columns=other.columns, method=...)

pandas.DataFrame.rename

DataFrame.rename(index=None, columns=None, copy=True, inplace=False)

Alter index and / or columns using 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.

Parameters
- index : dict-like or function, optional
  Transformation to apply to index values
- columns : dict-like or function, optional
  Transformation to apply to column values
- 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.

Returns
- renamed : DataFrame (new object)

See Also:
Series.rename
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

pandas.DataFrame.select

DataFrame.select (crit, axis=0)
Return data corresponding to axis labels matching criteria

Parameters
crit : function
    To be called on each index (label). Should return True or False

axis : int

Returns selection : type of caller

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

>>> indexed_df = df.set_index(['A', 'B'])
>>> indexed_df2 = df.set_index(['A', [0, 1, 2, 0, 1, 2]])
>>> indexed_df3 = df.set_index([[0, 1, 2, 0, 1, 2]])

pandas.DataFrame.tail

DataFrame.tail(n=5)

Returns last n rows of DataFrame

pandas.DataFrame.take

DataFrame.take(indices, axis=0)

Analogous to ndarray.take, return DataFrame corresponding to requested indices along an axis

Parameters  indices : list / array of ints

axis : {0, 1}

Returns  taken : DataFrame

pandas.DataFrame.truncate

DataFrame.truncate(before=None, after=None, copy=True)

Function truncate a sorted DataFrame / Series before and/or after some particular dates.

Parameters  before : date

Truncate before date

after : date

Truncate after date

Returns  truncated : type of caller

21.3.8 Missing data handling

pandas.DataFrame.dropna

DataFrame.dropna(axis=0, how='any', thresh=None, subset=None)

Return object with labels on given axis omitted where alternately any or all of the data are missing
**Parameters**

- `axis`: `{0, 1}`, or tuple/list thereof
  
  Pass tuple or list to drop on multiple axes

- `how`: `{'any', 'all'}`
  
  - any: if any NA values are present, drop that label
  - all: if all values are NA, drop that label

- `thresh`: int, default None
  
  int value: require that many non-NA values

- `subset`: array-like
  
  Labels along other axis to consider, e.g. if you are dropping rows these would be a list of columns to include

**Returns**

- `dropped`: DataFrame

---

**pandas.DataFrame.fillna**

**DataFrame.fillna** *(value=None, method=None, axis=0, inplace=False, limit=None)*

Fill NA/NaN values using the specified method

**Parameters**

- `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

- `value`: scalar or dict
  
  Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled)

- `axis`: `{0, 1}`, default 0
  
  - 0: fill column-by-column
  - 1: fill row-by-row

- `inplace`: boolean, default False
  
  If True, fill the DataFrame in place. Note: this will modify any other views on this DataFrame, like if you took a no-copy slice of an existing DataFrame, for example a column in a DataFrame. Returns a reference to the filled object, which is self if inplace=True

- `limit`: int, default None
  
  Maximum size gap to forward or backward fill

**Returns**

- `filled`: DataFrame

**See Also**

- `reindex`, `asfreq`

---

### 21.3.9 Reshaping, sorting, transposing

- `DataFrame.delevel(*args, **kwargs)`
- `DataFrame.pivot([index, columns, values])` Reshape data (produce a “pivot” table) based on column values.
- `DataFrame.reorder_levels(order[, axis])` Rearrange index levels using input order.

---

21.3. DataFrame 415
Table 21.30 – continued from previous page

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.sort</td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td>DataFrame.sort_index</td>
<td>Sort DataFrame either by labels (along either axis) or by the values in</td>
</tr>
<tr>
<td>DataFrame.sortlevel</td>
<td>Sort multilevel index by chosen axis and primary level.</td>
</tr>
<tr>
<td>DataFrame.swaplevel</td>
<td>Swap levels i and j in a MultiIndex on a particular axis</td>
</tr>
<tr>
<td>DataFrame.stack</td>
<td>Pivot a level of the (possibly hierarchical) column labels, returning a</td>
</tr>
<tr>
<td>DataFrame.unstack</td>
<td>Pivot a level of the (necessarily hierarchical) index labels, returning</td>
</tr>
<tr>
<td>DataFrame.to_panel</td>
<td>Returns a DataFrame with the rows/columns switched. If the DataFrame is</td>
</tr>
<tr>
<td>DataFrame.transpose</td>
<td>Transform long (stacked) format (DataFrame) into wide (3D, Panel)</td>
</tr>
<tr>
<td>DataFrame.delevel</td>
<td>Returns a DataFrame with the rows/columns switched. If the DataFrame is</td>
</tr>
<tr>
<td>DataFrame.pivot</td>
<td>Reshape data (produce a “pivot” table) based on column values. Uses unique</td>
</tr>
<tr>
<td></td>
<td>values from index / columns to form axes and return either DataFrame or</td>
</tr>
<tr>
<td></td>
<td>Panel, depending on whether you request a single value column (DataFrame)</td>
</tr>
<tr>
<td></td>
<td>or all columns (Panel)</td>
</tr>
<tr>
<td>Parameters</td>
<td>index : string or object</td>
</tr>
<tr>
<td></td>
<td>Column name to use to make new frame’s index</td>
</tr>
<tr>
<td>columns : string or object</td>
<td>Column name to use to make new frame’s columns</td>
</tr>
<tr>
<td>values : string or object</td>
<td>Column name to use for populating new frame’s values</td>
</tr>
<tr>
<td>Returns</td>
<td>pivoted : DataFrame</td>
</tr>
<tr>
<td></td>
<td>If no values column specified, will have hierarchically indexed columns</td>
</tr>
</tbody>
</table>

Notes

For finer-tuned control, see hierarchical indexing documentation along with the related stack/unstack methods

Examples

```python
>>> df
  foo  bar  baz
0   A  1.
1   B  2.
2   C  3.
3   A  4.
4   B  5.
5   C  6.
```
```python
>>> df.pivot('foo', 'bar', 'baz')
   A   B   C
one 1   2   3
two 4   5   6
```

```python
>>> df.pivot('foo', 'bar')['baz']
   A   B   C
one 1   2   3
two 4   5   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 representing new level order. :  
|-------|-------------------------------------------------  

(order by number not by key)  

| axis | where to reorder levels :  
|------|-------------------------------------------------  

| Returns | type of caller (new object) :  
|--------|---------------------------------------------------------------------  

### pandas.DataFrame.sort

**DataFrame.sort**(columns=None, column=None, axis=0, ascending=True, inplace=False)  
Sort DataFrame either by labels (along either axis) or by the values in column(s)  

**Parameters**  

| columns | object  
|---------|-------------------------------------------------  

(Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.)  

| ascending | boolean or list, default True  
|-----------|-------------------------------------------------  

(Sort ascending vs. descending. Specify list for multiple sort orders.)  

| axis | {0, 1}  
|-----|-------------------------------------------------  

(Sort index/rows versus columns.)  

| inplace | boolean, default False  
|---------|-------------------------------------------------  

(Sort the DataFrame without creating a new instance.)  

| Returns | sorted : DataFrame  
|--------|-------------------------------------------------  

**Examples**

```python
>>> result = df.sort(['A', 'B'], ascending=[1, 0])
```

### pandas.DataFrame.sort_index

**DataFrame.sort_index**(axis=0, by=None, ascending=True, inplace=False)  
Sort DataFrame either by labels (along either axis) or by the values in a column  

**Parameters**

| axis | {0, 1}  
|-----|-------------------------------------------------  

(Sort index/rows versus columns.)
by : object

Column name(s) in frame. Accepts a column name or a list or tuple for a nested sort.

ascending : boolean or list, default True

Sort ascending vs. descending. Specify list for multiple sort orders

inplace : boolean, default False

Sort the DataFrame without creating a new instance

Returns sorted : DataFrame

Examples

```python
gresult = df.sort_index(by=[‘A’, ‘B’], ascending=[1, 0])
```

pandas.DataFrame.sortlevel

DataFrame.sortlevel (level=0, axis=0, ascending=True, inplace=False)

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)

Parameters level : int

axis : {0, 1}

ascending : bool, default True

inplace : boolean, default False

Sort the DataFrame without creating a new instance

Returns sorted : DataFrame

pandas.DataFrame.swaplevel

DataFrame.swaplevel (i, j, 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)

pandas.DataFrame.stack

DataFrame.stack (level=-1, dropna=True)

Pivot a level of the (possibly hierarchical) column labels, returning a DataFrame (or Series in the case of an object with a single level of column labels) having a hierarchical index with a new inner-most level of row labels.

Parameters level : int, string, or list of these, default last level

Level(s) to stack, can pass level name

dropna : boolean, default True
Whether to drop rows in the resulting Frame/Series with no valid values

**Returns** stacked : DataFrame or Series

### Examples

```python
going s
   a  b
one 1. 2.
two 3. 4.

>>> s.stack()
   one  a  1
       b  2
   two a  3
       b  4
```

### pandas.DataFrame.unstack

**DataFrame.unstack**(level=-1)

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)

**Parameters** level : int, string, or list of these, default last level

Level(s) of index to unstack, can pass level name

**Returns** unstacked : DataFrame or Series

### Examples

```python
going s
one a 1.
one b 2.
two a 3.
two b 4.

>>> s.unstack(level=-1)
   a  b
one 1. 2.
two 3. 4.

>>> df = s.unstack(level=0)

>>> df
   one two
   a 1. 2.
   b 3. 4.

>>> df.unstack()
   one a 1.
       b 3.
two a 2.
       b 4.
```
pandas.DataFrame.T

DataFrame.T
Returns a DataFrame with the rows/columns switched. If the DataFrame is homogeneously-typed, the data is not copied.

pandas.DataFrame.to_panel

DataFrame.to_panel()
Transform long (stacked) format (DataFrame) into wide (3D, Panel) format.
Currently the index of the DataFrame must be a 2-level MultiIndex. This may be generalized later

Returns  panel : Panel

pandas.DataFrame.transpose

DataFrame.transpose()
Returns a DataFrame with the rows/columns switched. If the DataFrame is homogeneously-typed, the data is not copied.

21.3.10 Combining / joining / merging

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataFrame.append</td>
<td>Append columns of other to end of this frame’s columns and index, returning a new object. Columns not in this frame are added as new columns.</td>
</tr>
<tr>
<td>DataFrame.join</td>
<td>Join columns with other DataFrame either on index or on a key</td>
</tr>
<tr>
<td>DataFrame.merge</td>
<td>Merge DataFrame objects by performing a database-style join operation by</td>
</tr>
<tr>
<td>DataFrame.replace</td>
<td>Replace values given in ‘to_replace’ with ‘value’ or using ‘method’</td>
</tr>
<tr>
<td>DataFrame.update</td>
<td>Modify DataFrame in place using non-NA values from passed</td>
</tr>
</tbody>
</table>

pandas.DataFrame.append

DataFrame.append(other[, ignore_index, ...])
Append columns of other to end of this frame’s columns and index, returning a new object. Columns not in this frame are added as new columns.

Parameters
- other : DataFrame or list of Series/dict-like objects
- ignore_index : boolean, default False
  If True do not use the index labels. Useful for gluing together record arrays
- verify_integrity : boolean, default False
  If True, raise Exception on creating index with duplicates

Returns  appended : DataFrame

Notes
If a list of dict 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
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 : column name, tuple/list of column names, or array-like

Column(s) to use for joining, otherwise join on index. If multiples columns given, the passed DataFrame must have a MultiIndex. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation

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

sort : boolean, default False

Order result DataFrame lexicographically by the join key. If False, preserves the index order of the calling (left) DataFrame

Returns

joined : DataFrame

Notes

on, lsuffix, and rsuffix options are not supported when passing a list of DataFrame objects

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)

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 (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)

**on**: label or list

Field names to join on. Must be found in both DataFrames. If on is None and not merging on indexes, then it merges on the intersection of the columns by default.

**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_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

**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

**Returns** merged : DataFrame

**Examples**

```python
>>> A
     lkey  value
  0   foo   1
  1   bar   2
  2   baz   3
  3   foo   4

>>> B
     rkey  value
  0   foo   5
  1   bar   6
  2   qux   7
  3   bar   8

>>> merge(A, B, left_on='lkey', right_on='rkey', how='outer')
     lkey  value_x  rkey  value_y
  0   bar     2     bar     6
  1   bar     2     bar     8
  2   baz     3       NaN      NaN
  3   foo     1     foo     5
  4   foo     4     foo     5
  5     NaN   NaN     qux     7
```

**pandas.DataFrame.replace**

DataFrame.replace(to_replace, value=None, method='pad', axis=0, inplace=False, limit=None)

Replace values given in ‘to_replace’ with ‘value’ or using ‘method’
Parameters  value : scalar or dict, default None

Value to use to fill holes (e.g. 0), alternately a dict of values specifying which value to use for each column (columns not in the dict will not be filled)

method : {'backfill', 'bfill', 'pad', 'ffill', None}, default 'pad'

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}, default 0

0: fill column-by-column 1: fill row-by-row

inplace : boolean, default False

If True, fill the DataFrame in place. Note: this will modify any other views on this DataFrame, like if you took a no-copy slice of an existing DataFrame, for example a column in a DataFrame. Returns a reference to the filled object, which is self if inplace=True

limit : int, default None

Maximum size gap to forward or backward fill

Returns  filled : DataFrame

See Also:

reindex, asfreq

pandas.DataFrame.update

DataFrame.update(other, join='left', overwrite=True, filter_func=None, raise_conflict=False)

Modify DataFrame in place using non-NA values from passed DataFrame. Aligns on indices

Parameters  other : DataFrame, or object coercible into a DataFrame

join : {'left', 'right', 'outer', 'inner'}, default 'left'

overwrite : boolean, default True

If True then overwrite values for common keys in the calling frame

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 the DataFrame and other both contain data in the same place

21.3.11 Time series-related

| pandas.DataFrame.asfreq(freq[, method, how, normalize]) | Convert all TimeSeries inside to specified frequency using DateOffset |
| pandas.DataFrame.shift([periods, freq]) | Shift the index of the DataFrame by desired number of periods with an |
| pandas.DataFrame.first_valid_index() | Return label for first non-NA/null value |
| pandas.DataFrame.last_valid_index() | Return label for last non-NA/null value |
**Table 21.32 – continued from previous page**

- **DataFrame.resample**(rule[, how, axis, ...]) Convenience method for frequency conversion and resampling of regular time-series data.
- **DataFrame.to_period**([freq, axis, copy]) Convert DataFrame from DatetimeIndex to PeriodIndex with desired
- **DataFrame.to_timestamp**([freq, how, axis, copy]) Cast to DatetimeIndex of timestamps, at *beginning* of period
- **DataFrame.tz_convert**(tz[, axis, copy]) Convert TimeSeries to target time zone. If it is time zone naive, it
- **DataFrame.tz_localize**(tz[, axis, copy]) Localize tz-naive TimeSeries to target time zone

### pandas.DataFrame.asfreq

**DataFrame.asfreq**(freq, method=None, how=None, normalize=False)
Convert all TimeSeries inside to specified frequency using DateOffset objects. Optionally provide fill method to pad/backfill missing values.

**Parameters**

- freq : DateOffset object, or string
- method : {'backfill', 'bfill', 'pad', 'ffill', 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 method
- how : {'start', 'end'}, default end
  For PeriodIndex only, see PeriodIndex.asfreq
- normalize : bool, default False
  Whether to reset output index to midnight

**Returns**
converted : type of caller

### pandas.DataFrame.shift

**DataFrame.shift**(periods=1, freq=None, **kwds)
Shift the index of the DataFrame 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 datetools module or time rule (e.g. ‘EOM’)

**Returns**
shifted : DataFrame

**Notes**

If freq is specified then the index values are shifted but the data if not realigned

### pandas.DataFrame.first_valid_index

**DataFrame.first_valid_index**()
Return label for first non-NA/null value
pandas.DataFrame.last_valid_index

DataFrame.last_valid_index()
Return label for last non-NA/null value

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)
Convenience method for frequency conversion and resampling of regular time-series data.

Parameters
- **rule**: the offset string or object representing target conversion
  - **how**: string, method for down- or re-sampling, default to ‘mean’ for downsampling
  - **axis**: int, optional, default 0
  - **fill_method**: string, fill_method for upsampling, default None
  - **closed**: {'right', 'left'}, default None
    - Which side of bin interval is closed
  - **label**: {'right', 'left'}, default None
    - Which bin edge label to label bucket with
  - **convention**: {'start', 'end', 's', 'e'}
  - **kind**: “period”/”timestamp”
  - **loffset**: timedelta
    - Adjust the resampled time labels
  - **limit**: int, default None
    - Maximum size gap to when reindexing with fill_method
  - **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

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, 1}, 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_timestamp**

```python
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, 1} 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.tz_convert**

```python
Dataframe.tz_convert(tz, axis=0, copy=True)
```

Convert TimeSeries to target time zone. If it is time zone naive, it will be localized to the passed time zone.

**Parameters**

- `tz`: string or pytz.timezone object
- `copy`: boolean, default True
  - Also make a copy of the underlying data

**pandas.DataFrame.tz_localize**

```python
Dataframe.tz_localize(tz, axis=0, copy=True)
```

Localize tz-naive TimeSeries to target time zone.

**Parameters**

- `tz`: string or pytz.timezone object
- `copy`: boolean, default True
  - Also make a copy of the underlying data

**21.3.12 Plotting**

**DataFrame.boxplot**

```python
DataFrame.boxplot([column, by, ax, ...])
```

Make a box plot from DataFrame column/columns optionally grouped.

**DataFrame.hist**

```python
DataFrame.hist(data[, column, by, grid, ...])
```

Draw Histogram the DataFrame’s series using matplotlib / pylab.

**DataFrame.plot**

```python
DataFrame.plot([frame, x, y, subplots, ...])
```

Make line or bar plot of DataFrame’s series with the index on the x-axis.

**pandas.DataFrame.boxplot**

```python
DataFrame.boxplot(column=None, by=None, ax=None, fontsize=None, rot=0, grid=True, **kwds)
```

Make a box plot from DataFrame column/columns optionally grouped (stratified) by one or more columns.

**Parameters**

- `data`: DataFrame
- `column`: column names or list of names, or vector

---

426 Chapter 21. API Reference
Can be any valid input to groupby

**by** : string or sequence
  Column in the DataFrame to group by

**fontsize** : int or string
  Returns ax : matplotlib.axes.AxesSubplot

### pandas.DataFrame.hist

DataFrame.hist(data=None, column=None, by=None, grid=True, xlabelsize=None, xrot=None, ylabelsize=None, yrot=None, ax=None, sharex=False, sharey=False, **kwds)

Draw Histogram the DataFrame’s series using matplotlib / pylab.

**Parameters**

**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

**ax** : matplotlib axes object, default None

**sharex** : bool, if True, the X axis will be shared amongst all subplots.

**sharey** : bool, if True, the Y axis will be shared amongst all subplots.

**kwds** : other plotting keyword arguments
  To be passed to hist function

### pandas.DataFrame.plot

DataFrame.plot(frame=None, x=None, y=None, subplots=False, sharex=True, sharey=False, use_index=True, figsize=None, grid=False, legend=True, rot=None, ax=None, style=None, title=None, xlim=None, ylim=None, logx=False, logy=False, xticks=None, yticks=None, kind='line', sort_columns=False, fontsize=None, secondary_y=False, **kwds)

Make line or bar plot of DataFrame’s series with the index on the x-axis using matplotlib / pylab.

**Parameters**

**x** : label or position, default None

**y** : label or position, default None
  Allows plotting of one column versus another

**subplots** : boolean, default False
  Make separate subplots for each time series
sharex : boolean, default True
   In case subplots=True, share x axis
sharey : boolean, default False
   In case subplots=True, share y axis
use_index : boolean, default True
   Use index as ticks for x axis
stacked : boolean, default False
   If True, create stacked bar plot. Only valid for DataFrame input
sort_columns: boolean, default False :
   Sort column names to determine plot ordering
title : string
   Title to use for the plot
grid : boolean, default False
   Axis grid lines
legend : boolean, default True
   Place legend on axis subplots
ax : matplotlib axis object, default None
style : list or dict
   matplotlib line style per column
   bar : vertical bar plot barh : horizontal bar plot kde/density : Kernel Density Estimation plot
logx : boolean, default False
   For line plots, use log scaling on x axis
logy : boolean, default False
   For line plots, use log scaling on y axis
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
secondary_y : boolean or sequence, default False
   Whether to plot on the secondary y-axis If dict then can select which columns to plot on secondary y-axis
kws : keywords
    Options to pass to matplotlib plotting method

Returns ax_or_axes : matplotlib.AxesSubplot or list of them

21.3.13 Serialization / IO / Conversion

DataFrame.from_csv(path[, header, sep, ...])          Read delimited file into DataFrame
DataFrame.from_dict(data[, orient, dtype])          Construct DataFrame from dict of array-like or dicts
DataFrame.from_items(items[, columns, orient])      Convert (key, value) pairs to DataFrame. The keys will be the axis
DataFrame.from_records(data[, index, ...])          Convert structured or record ndarray to DataFrame
DataFrame.info([verbose, buf, max_cols])            Concise summary of a DataFrame, used in __repr__ when very large.
DataFrame.load(path)                                Load DataFrame
DataFrame.save(path)                                 Save DataFrame
DataFrame.to_csv(path_or_buf[, sep, na_rep, ...])    Write DataFrame to a comma-separated values (csv) file
DataFrame.to_dict([orient])                         Convert DataFrame to dictionary.
DataFrame.to_excel(excel_writer[, ...])            Write DataFrame to a excel sheet
DataFrame.to_html([buf, columns, col_space, ...])  to_html-specific options
DataFrame.to_records([index, convert_datetime64])  Convert DataFrame to record array. Index will be put in the
DataFrame.to_sparse([fill_value, kind])             Convert to SparseDataFrame
DataFrame.to_string([buf, columns, ...])           Render a DataFrame to a console-friendly tabular output

pandas.DataFrame.from_csv

classmethod DataFrame.from_csv(path[, header=0, sep=',', na_rep=None, index_col=0, parse_dates=True, encoding=None])

Read delimited file into DataFrame

Parameters

path : string file path or file handle / StringIO

header : int, default 0
    Row to use at 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

Returns

y : DataFrame

Notes

Preferable to use read_table for most general purposes but from_csv makes for an easy roundtrip to and from file, especially with a DataFrame of time series data
pandas: powerful Python data analysis toolkit, Release 0.11.0.dev-9988e5f

pandas.DataFrame.from_dict

classmethod DataFrame.from_dict(data, orient='columns', dtype=None)

Construct DataFrame from dict of array-like or dicts

Parameters:
- data : dict
  {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’.

Returns:
- DataFrame

pandas.DataFrame.from_items

classmethod DataFrame.from_items(items, columns=None, orient='columns')

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 named 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 to non-string, non-numeric objects (like decimal.Decimal) to floating point, useful for SQL result sets

Returns df : DataFrame

pandas.DataFrame.info

DataFrame.info (verbose=True, buf=None, max_cols=None)
Concise summary of a DataFrame, used in __repr__ when very large.

Parameters verbose : boolean, default True
If False, don’t print column count summary
buf : writable buffer, defaults to sys.stdout
max_cols : int, default None
 Determines whether full summary or short summary is printed

pandas.DataFrame.load

classmethod DataFrame.load (path)

pandas.DataFrame.save

DataFrame.save (path)

pandas.DataFrame.to_csv

DataFrame.to_csv (path_or_buf, sep=',', na_rep='', float_format=None, cols=None, header=True,
index=True, index_label=None, mode='w', encoding=None, quoting=None, line_terminator=None)
Write DataFrame to a comma-separated values (csv) file

Parameters path_or_buf : string or file handle / StringIO
 File path
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
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, 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
nanRep : deprecated, use na_rep mode : Python write mode, default ‘w’ encoding :
string, optional
    a string representing the encoding to use if the contents are non-ascii, for python
versions prior to 3
line_terminator: string, default ‘
’
    The newline character or character sequence to use in the output file
quoting [optional constant from csv module] defaults to csv.QUOTE_MINIMAL

pandas.DataFrame.to_dict

DataFrame.to_dict (outtype=’dict’)
    Convert DataFrame to dictionary.

    Parameters outtype : str {‘dict’, ‘list’, ‘series’}
    Determines the type of the values of the dictionary. The default dict is a nested dictio-
nary {column -> {index -> value}}, list returns {column -> list(values)}, series returns
{column -> Series(values)}. Abbreviations are allowed.

    Returns result : dict like {column -> {index -> value}}

pandas.DataFrame.to_excel

DataFrame.to_excel (excel_writer, sheet_name=’sheet1’, na_rep=’’, float_format=None, cols=None,
header=True, index=True, index_label=None, startrow=0, startcol=0)
    Write DataFrame to a 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
    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

432 Chapter 21. API Reference
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

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 = ExcelWriter('output.xlsx') >>> df1.to_excel(writer,'sheet1') >>> df2.to_excel(writer,'sheet2') >>> writer.save()

pandas.DataFrame.to_html

DataFrame.to_html (buf=None, columns=None, col_space=None, colSpace=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, index_names=True, justify=None, force_unicode=None, bold_rows=True, classes=None)
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

Render a DataFrame to an html table.

Parameters frame : DataFrame

object to render

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, if the result is a string, it 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

**sparsify** : bool, optional
Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

**justify** : {'left', 'right'}, default None
Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_printoptions), ‘right’ out of the box.

**index_names** : bool, optional
Prints the names of the indexes, default True

**force_unicode** : bool, default False
Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

**Returns**  **formatted** : string (or unicode, depending on data and options)

### pandas.DataFrame.to_records

DataFrame.to_records (index=True, convert_datetime64=True)
Convert DataFrame to 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 True
Whether to convert the index to datetime.datetime if it is a DatetimeIndex

**Returns**  **y** : recarray

### 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_string

DataFrame.to_string (buf=None, columns=None, col_space=None, colSpace=None, header=True, index=True, na_rep='NaN', formatters=None, float_format=None, sparsify=None, nanRep=None, index_names=True, justify=None, force_unicode=None, line_width=None)
Render a DataFrame to a console-friendly tabular output.

**Parameters**  **frame** : DataFrame
object to render

**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, if the result is a string, it 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

**sparsify** : bool, optional

Set to False for a DataFrame with a hierarchical index to print every multiindex key at each row, default True

**justify** : {'left', 'right'}, default None

Left or right-justify the column labels. If None uses the option from the print configuration (controlled by set_printoptions), ‘right’ out of the box.

**index_names** : bool, optional

Prints the names of the indexes, default True

**force_unicode** : bool, default False

Always return a unicode result. Deprecated in v0.10.0 as string formatting is now rendered to unicode by default.

**Returns**

**formatted** : string (or unicode, depending on data and options)

## 21.4 Panel

### 21.4.1 Computations / Descriptive Stats
pandas, 1
pandas, 1
Symbols

_init__() (pandas.DataFrame method), 387
_init__() (pandas.Series method), 355
__iter__() (pandas.DataFrame method), 389
__iter__() (pandas.Series method), 356

A

abs() (pandas.DataFrame method), 398
abs() (pandas.Series method), 361
add() (pandas.DataFrame method), 391
add() (pandas.Series method), 357
add_prefix() (pandas.DataFrame method), 407
add_suffix() (pandas.DataFrame method), 407
align() (pandas.DataFrame method), 407
align() (pandas.Series method), 369
any() (pandas.DataFrame method), 398
any() (pandas.Series method), 361
append() (pandas.DataFrame method), 420
append() (pandas.Series method), 377
apply() (pandas.DataFrame method), 396
apply() (pandas.Series method), 359
applymap() (pandas.DataFrame method), 397
argsort() (pandas.Series method), 375
as_matrix() (pandas.DataFrame method), 385
asfreq() (pandas.DataFrame method), 424
asfreq() (pandas.Series method), 379
asof() (pandas.Series method), 379
astype() (pandas.DataFrame method), 388
astype() (pandas.Series method), 355
autocorr() (pandas.Series method), 361
axes (pandas.DataFrame attribute), 386

B

between() (pandas.Series method), 362
boxplot() (pandas.DataFrame method), 426

c

clip() (pandas.DataFrame method), 399
clip() (pandas.Series method), 362
clip_lower() (pandas.DataFrame method), 399
clip_lower() (pandas.Series method), 362
clip_upper() (pandas.DataFrame method), 399
clip_upper() (pandas.Series method), 362
combine() (pandas.DataFrame method), 395
combine() (pandas.Series method), 358
combine_first() (pandas.DataFrame method), 395
combine_first() (pandas.Series method), 358
combineAdd() (pandas.DataFrame method), 395
combineMult() (pandas.DataFrame method), 396
concat() (in module pandas.tools.merge), 333
cov() (pandas.DataFrame method), 400
cov() (pandas.Series method), 363
cummax() (pandas.DataFrame method), 400
cummax() (pandas.Series method), 363
cummin() (pandas.DataFrame method), 401
cummin() (pandas.Series method), 363
cumprod() (pandas.DataFrame method), 401
cumprod() (pandas.Series method), 364
cumsum() (pandas.DataFrame method), 401
cumsum() (pandas.Series method), 364

d

delevel() (pandas.DataFrame method), 416
describe() (pandas.DataFrame method), 401
describe() (pandas.Series method), 364
diff() (pandas.DataFrame method), 402
diff() (pandas.Series method), 364
div() (pandas.DataFrame method), 392
div() (pandas.Series method), 357
drop() (pandas.DataFrame method), 408
drop() (pandas.Series method), 370
drop_duplicates() (pandas.DataFrame method), 408
dropna() (pandas.DataFrame method), 414
dropna() (pandas.Series method), 374
dtype (pandas.Series attribute), 354
Index

dtypes (pandas.DataFrame attribute), 386
duplicated() (pandas.DataFrame method), 409

E

ewma() (in module pandas.stats.moments), 350
ewmccorr() (in module pandas.stats.moments), 352
ewmcov() (in module pandas.stats.moments), 353
ewmstd() (in module pandas.stats.moments), 351
ewmvar() (in module pandas.stats.moments), 352
expanding_apply() (in module pandas.stats.moments), 349
expanding_corr() (in module pandas.stats.moments), 348
expanding_count() (in module pandas.stats.moments), 346
expanding_cov() (in module pandas.stats.moments), 348
expanding_kurt() (in module pandas.stats.moments), 349
expanding_mean() (in module pandas.stats.moments), 347
expanding_median() (in module pandas.stats.moments), 347
expanding_quantile() (in module pandas.stats.moments), 350
expanding_skew() (in module pandas.stats.moments), 349
expanding_sum() (in module pandas.stats.moments), 347
expanding_var() (in module pandas.stats.moments), 347

F

fillna() (pandas.DataFrame method), 415
fillna() (pandas.Series method), 374
filter() (pandas.DataFrame method), 409
first() (pandas.DataFrame method), 409
first() (pandas.Series method), 370
first_valid_index() (pandas.DataFrame method), 424
first_valid_index() (pandas.Series method), 380
from_csv() (pandas.DataFrame class method), 429
from_csv() (pandas.Series class method), 383
from_dict() (pandas.DataFrame class method), 430
from_items() (pandas.DataFrame class method), 430
from_records() (pandas.DataFrame class method), 430

G

get() (pandas.io.pytables.HDFStore method), 341
get() (pandas.Series method), 356
get_dtype_counts() (pandas.DataFrame method), 386
groupby() (pandas.DataFrame method), 397
groupby() (pandas.Series method), 360

H

head() (pandas.DataFrame method), 388, 410
head() (pandas.Series method), 370
hist() (pandas.DataFrame method), 427
hist() (pandas.Series method), 381

I

idxmax() (pandas.DataFrame method), 410
idxmax() (pandas.Series method), 370
idxmin() (pandas.DataFrame method), 410
idxmin() (pandas.Series method), 371
info() (pandas.DataFrame method), 431
insert() (pandas.DataFrame method), 389
interpolate() (pandas.Series method), 375
isinf() (pandas.Series method), 371
isnan() (pandas.Series method), 355
iteritems() (pandas.DataFrame method), 389
iteritems() (pandas.Series method), 356
iterrows() (pandas.DataFrame method), 389
itertuples() (pandas.DataFrame method), 389
ix (pandas.DataFrame attribute), 389
ix (pandas.Series attribute), 356

J

join() (pandas.DataFrame method), 421

K

kurt() (pandas.DataFrame method), 402
kurt() (pandas.Series method), 364

L

last() (pandas.DataFrame method), 410
last() (pandas.Series method), 371
last_valid_index() (pandas.DataFrame method), 425
last_valid_index() (pandas.Series method), 380
load() (in module pandas.core.common), 334
load() (pandas.DataFrame class method), 431
load() (pandas.Series class method), 383
lookup() (pandas.DataFrame method), 389

M

mad() (pandas.DataFrame method), 402
mad() (pandas.Series method), 365
map() (pandas.Series method), 359
max() (pandas.DataFrame method), 402
max() (pandas.Series method), 365
mean() (pandas.DataFrame method), 403
mean() (pandas.Series method), 365
median() (pandas.DataFrame method), 403
median() (pandas.Series method), 366
merge() (in module pandas.tools.merge), 332
merge() (pandas.DataFrame method), 421
min() (pandas.DataFrame method), 403
min() (pandas.Series method), 366
mul() (pandas.DataFrame method), 392
mul() (pandas.Series method), 357
N
ndim (pandas.DataFrame attribute), 386
notnull() (pandas.Series method), 355
nunique() (pandas.Series method), 366

O
order() (pandas.Series method), 375

P
pandas (module), 1
parse() (pandas.io.parsers.ExcelFile method), 340
pct_change() (pandas.DataFrame method), 404
pct_change() (pandas.Series method), 366
pivot() (pandas.DataFrame method), 416
pivot_table() (in module pandas.tools.pivot), 331
plot() (pandas.DataFrame method), 427
plot() (pandas.Series method), 382
pop() (pandas.DataFrame method), 390
prod() (pandas.DataFrame method), 404
prod() (pandas.Series method), 367
put() (pandas.io.pytables.HDFStore method), 341

Q
quantile() (pandas.DataFrame method), 404
quantile() (pandas.Series method), 367

R
radd() (pandas.DataFrame method), 393
rank() (pandas.DataFrame method), 405
rank() (pandas.Series method), 367
rdiv() (pandas.DataFrame method), 394
read_csv() (in module pandas.io.parsers), 338
read_table() (in module pandas.io.parsers), 335
reindex() (pandas.DataFrame method), 410
reindex() (pandas.Series method), 371
reindex_axis() (pandas.DataFrame method), 411
reindex_like() (pandas.DataFrame method), 412
reindex_like() (pandas.Series method), 372
rename() (pandas.DataFrame method), 412
rename() (pandas.Series method), 372
reorder_levels() (pandas.DataFrame method), 417
reorder_levels() (pandas.Series method), 376
replace() (pandas.DataFrame method), 422
replace() (pandas.Series method), 378
resample() (pandas.DataFrame method), 425
resample() (pandas.Series method), 380
reset_index() (pandas.DataFrame method), 413
reset_index() (pandas.Series method), 373
rmul() (pandas.DataFrame method), 394
rolling_apply() (in module pandas.stats.moments), 345
rolling_apply() (pandas.DataFrame method), 394
rolling_corr() (in module pandas.stats.moments), 344
rolling_count() (in module pandas.stats.moments), 342
rolling_cov() (in module pandas.stats.moments), 344
rolling_kurt() (in module pandas.stats.moments), 345
rolling_mean() (in module pandas.stats.moments), 342
rolling_median() (in module pandas.stats.moments), 343
rolling_quantile() (in module pandas.stats.moments), 346
rolling_skew() (in module pandas.stats.moments), 343
rolling_std() (in module pandas.stats.moments), 344
rolling_sum() (in module pandas.stats.moments), 342
rolling_var() (in module pandas.stats.moments), 343
round() (pandas.Series method), 358
rsub() (pandas.DataFrame method), 395

S
save() (in module pandas.core.common), 335
save() (pandas.DataFrame method), 431
save() (pandas.Series method), 384
select() (pandas.DataFrame method), 413
select() (pandas.Series method), 373
set_index() (pandas.DataFrame method), 413
shape (pandas.DataFrame attribute), 386
shift() (pandas.DataFrame method), 424
shift() (pandas.Series method), 379
skew() (pandas.DataFrame method), 405
skew() (pandas.Series method), 367
sort() (pandas.DataFrame method), 417
sort() (pandas.Series method), 376
sort_index() (pandas.DataFrame method), 417
sort_index() (pandas.Series method), 376
sortlevel() (pandas.DataFrame method), 418
sortlevel() (pandas.Series method), 376
stack() (pandas.DataFrame method), 418
std() (pandas.DataFrame method), 406
std() (pandas.Series method), 368
sub() (pandas.DataFrame method), 393
sub() (pandas.Series method), 358
sum() (pandas.DataFrame method), 405
sum() (pandas.Series method), 368
swaplevel() (pandas.DataFrame method), 418
swaplevel() (pandas.Series method), 377

T
t (pandas.DataFrame attribute), 420
tail() (pandas.DataFrame method), 390, 414
tail() (pandas.Series method), 373
take() (pandas.DataFrame method), 414
take() (pandas.Series method), 373
to_csv() (pandas.DataFrame method), 431
to_csv() (pandas.Series method), 384
to_dict() (pandas.DataFrame method), 432
to_dict() (pandas.Series method), 384
to_excel() (pandas.DataFrame method), 432
to_html() (pandas.DataFrame method), 433
to_panel() (pandas.DataFrame method), 420
to_period() (pandas.DataFrame method), 425
to_records() (pandas.DataFrame method), 434
to_sparse() (pandas.DataFrame method), 434
to_sparse() (pandas.Series method), 384
to_string() (pandas.DataFrame method), 434
to_string() (pandas.Series method), 385
to_timestamp() (pandas.DataFrame method), 426
transpose() (pandas.DataFrame method), 420
truncate() (pandas.DataFrame method), 414
truncate() (pandas.Series method), 374
tz_convert() (pandas.DataFrame method), 426
tz_convert() (pandas.Series method), 381
tz_localize() (pandas.DataFrame method), 426
tz_localize() (pandas.Series method), 381

U
unique() (pandas.Series method), 368
unstack() (pandas.DataFrame method), 419
unstack() (pandas.Series method), 377
update() (pandas.DataFrame method), 423
update() (pandas.Series method), 378

V
value_counts() (pandas.Series method), 369
values (pandas.DataFrame attribute), 386
values (pandas.Series attribute), 354
var() (pandas.DataFrame method), 406
var() (pandas.Series method), 368

W
weekday (pandas.Series attribute), 380

X
xs() (pandas.DataFrame method), 390