1 More pandas Functionality

1.1 Time-series Data

A `TimeSeries` is basically a series where the index contains datetimes index values (more formally the class TimeSeries inherits from Series), and Series constructor will automatically promote a Series with datetime index values to a TimeSeries. The TimeSeries examples all make use of US real GDP data from the Federal Reserve Economic Database (FRED).

The pandas `TimeSeries` object is currently limited to a span of about 585 years centered at 1970. While this is unlikely to create problems, it may not be appropriate for some applications. pandas includes a substantial number of routines which are primarily designed to work with timeseries data.

```
>>> GDP_data = read_excel('GDP.xls','GDP',skiprows=19)
>>> GDP_data.head()
>>> type(GDP_data.VALUE)
>>> gdp = GDP_data.VALUE
>>> gdp.index = GDP\_data.DATE
>>> gdp.head()
>>> type(gdp.index)
```

TimeSeries have some useful indexing tricks. For example, all of the data for a particular year can retrieved using `gdp['yyyy']` syntax where yyyy is a year.

```
>>> gdp['2009']
>>> gdp['200904']
          >>> gdp['2009':'2010']
>>> gdp['20090601':'20100601']
```

Slicing indexing can also be accomplished using datetime, for example `gdp[ datetime(2009,01,01): datetime(2011,12,31)]` where datetime has been imported using from pandas `import datetime`.

1.2 `date_range`

`date_range` is a very useful function provided by pandas to generate ranges of dates (from pandas `import date_range`). The basic use is either `date_range(beginning_date,ending_date)`

which will produce a daily series between the two dates (inclusive) or `date_range(beginning_date, periods=periods)` which will produce a daily series starting at `beginning_date` with periods periods.

```
>>> from pandas import date\_range
>>> date\_range('20130103','
20130105')
<class 'pandas.tseries.index.DatetimeIndex'>
[20130103
00:00:00, ..., 20130105
00:00:00]
Length: 3, Freq: D, Timezone: None
>>> date\_range('20130103',
periods = 3)
<class 'pandas.tseries.index.DatetimeIndex'>
[20130103
00:00:00, ..., 20130105
00:00:00]
Length: 3, Freq: D, Timezone: None
```

The keyword argument freq changes the frequency, and common choices include Scaling the frequency produces skips that are a multiple of the default, such as in 2D which uses every other day. Combining multiple frequencies produces less regular skips, e.g. 2H10T.

```
>>> date\_range('20130103', periods=4, freq='Q').values
array(['20130331T00: 00:00.000000000+0000',
'20130630T01:
00:00.000000000+0100',
'20130930T01:
00:00.000000000+0100',
'20131231T00:
00:00.000000000+0000'], dtype='datetime64[ns]')
>>> date\_range('20130103',
periods=4, freq='7D4H').values
array(['20130103T00:
00:00.000000000+0000',
'20130110T04:
00:00.000000000+0000',
```

```
'20130117T08:
00:00.000000000+0000',
'20130124T12:
00:00.000000000+0000'], dtype='datetime64[ns]')
```

Note that the underlying array uses NumPy's datetime64 as the data type (with nano-second resolution, indicated by [ns]).

resample

pandas supports sophisticated resampling which is useful for aggregating form a higher frequency to a lower one using resample. This example uses annual ('A') and alternative aggregation functions.

```
>>> gdp.resample('A',how=mean).tail() # Annual average
DATE
20091231
14417.950
20101231
14958.300
20111231
15533.825
20121231
16244.575
20131231
16601.600
Freq: ADEC,
dtype: float64
>>> gdp.resample('A',how=max).tail() # Maximum
DATE
20091231
14564.1
20101231
15231.7
20111231
15818.7
20121231
16420.3
20131231
16667.9
```

```
Freq: ADEC,
dtype: float64
```

pct_change

Growth rates are computed using `pct_change`. The keyword argument periods constructs overlapping growth rates which are useful when using seasonal data.

```
>>> gdp.pct\_change().tail()
DATE
20120401
0.007406
20120701
0.012104
20121001
0.003931
20130101
0.007004
20130401
0.008019
Name: VALUE, dtype: float64
>>> gdp.pct\_change(periods=4).tail() # Quarterly data, annual difference
DATE
20120401
0.045176
20120701
0.047669
20121001
0.038031
20130101
0.030776
20130401
0.031404
Name: VALUE, dtype: float64
```

1.3 Importing and Exporting Data

In addition to providing data management tools, pandas also excels at importing and exporting data. pandas supports reading and Excel, csv and other delimited files, Stata

files, fixed-width text, html, json, HDF5 and from SQL databases. The functions to read follow the common naming convention `read_type` where type is the file type, e.g. excel or csv. The writers are all methods of Series or DataFrame and follow the naming convention `to_type`.

1.3.1 Reading Data

read_excel
`read_excel` supports reading data fromboth xls (Excel 2003) and xlsx (Excel 2007/10/13) formats. The basic usage required two inputs, the file name and the sheet name. Other notable keyword arguments include:

- `header`, an integer indicating which row to use for the column labels. The default is 0 (top) row, and if skiprows is used, this value is relative.
- `skiprows`, typically an integer indicating the number of rows at the top of the sheet to skip before reading the file. The default is 0.
- `skip_footer`, typically an integer indicating the number of rows at the bottom of the sheet to skip when reading the file. The default is 0.
- `index_col`, an integer or column name indicating the column to use as the index. If not provided, a basic numeric index is generated.
- parse_cols, None, an integer, a list of integers or strings, tells pandas whether to attempt to parse a column. The default is `None` which will parse all columns. Alternatively, if an integer is provided then the value is interpreted as the last column to parse.
- Finally, if a list of integers is provided, the values are interpreted as the columns to parse (0-based, e.g. `[0,2,5]`).
- The string version takes one of the forms 'A', 'A,C,D', 'A:D' or a mix of the latter two ('A,C:D,G,W:Z').

read_csv
`read_csv` reads comma separated value files. The basic use only requires one input, a file name. `read_csv` also accepts valid URLs (http, ftp, or s3 (Amazon) if the boto package is available) or any object that provides a read method in places of the file name. A huge range of options are available, and so only the most relevant are presented in the list below.

- delimiter, the delimiter used to separate values. The default is ','. Complicated delimiters are matched using a regular expression.
- delim_whitespace, Boolean indicating that the delimiter is white space (a space or tab). This is preferred to using a regular expression to detect white space.

- header, an integer indicating the row number to use for the column names. The default is 0.
- skiprows, similar to skiprows in read_excel.
- skip_footer, similar to skip_footer in read_excel.
- index_col, similar to index_col in read_excel.
- names, a list of column names to use in-place of any found in the fileMust use header=0 (the default value).
- parse_dates, either a Boolean indicating whether to parse dates encountered, or a list of integers or strings indicating which columns to parse. Supportsmore complicated options to combine columns.
- date_parser, a function to use when parsing dates. The default parser is dateutil.parser.
- dayfirst, a Boolean indicating whether to use European date format (DD/MM, True) or American date format (MM/DD False) when encountering dates. The default is False.
- error_bad_lines, when True stops processing on a bad line. If False, continues skipping any bad lines encountered.
- encoding, a string containing the file encoding (e.g. 'utf8' or 'latin1').
- converters, a dictionary of functions for converting values in certain columns, where keys can either integers (column-number) or column labels.
- nrows, an integer, indicates the maximum number of rows to read. This is useful for reading a subset of a file.
- usecols, a list of integers or column names indicating which column to retain
- dtype A data type to use for the read data or a dictionary of data types using the column names as keys. If not provided, the type is inferred.

read_table
read_table is similar to read_csv and both are wrappers around a private read function provided by pandas. **read_hdf**
read_hdf is primarily for reading pandas DataTables which were written using DataTable.`to_hdf`

1.3.2 Writing Data

Writing data froma Series or DataFrame is muc simpler since the starting point (the Series or theDataFrame) is well understood by pandas. While the file writing methods all have a number of options, most can safely be ignored.

```
>>> state\_gdp.to\_excel('state\_gdp\_from\_dataframe.xls')
>>> state\_gdp.to\_excel('state\_gdp\_from\_dataframe\_sheetname.xls', sheet\_name='Sta
>>> state\_gdp.to\_excel('state\_gdp\_from\_dataframe.xlsx')
>>> state\_gdp.to\_csv('state\_gdp\_from\_dataframe.csv')
>>> import cStringIO
>>> sio = cStringIO.StringIO()
>>> state\_gdp.to\_json(sio)
>>> sio.seek(0)
>>> sio.read(50)
'{"state\_code":{"0":"AK","1":"AL","2":"AR","3":"AZ"'
>>> state\_gdp.to\_string()[:50]
u' state\_code state gdp\_2009 gdp'
```

One writer, `to_hdf` is worth special mention. to_hdf writes pandasDataFrames to-HDF5files which are binary files which support compression. HDF5 files can achieve fantastic compression ratios when data are regular, and so are often much more useful than csv or xlsx (which is also compressed). The usage of to_hdf is not meaningfully different from the other writers except that:

- In addition to the filename, an argument is required containing the key, which is usually the variable name.
- Two additional arguments must be passed for the output file to be compressed. These two keyword arguments are complib and complevel, which I recommend to setting to 'zlib' and 6, respectively.

```
>>> df = DataFrame(zeros((1000,1000)))
>>> df.to\_csv('size\_test.csv')
>>> df.to\_hdf('size\_test.h5','df') # h5 is the usual extension for HDF5
# h5 is the usual extension for HDF5
>>> df.to\_hdf('size\_test\_compressed.h5','df',complib='zlib',complevel=6)
>>> ls size\_* # Ignore
09/19/2013 04:16 PM 4,008,782 size\_test.csv
198
```

```
09/19/2013 04:16 PM 8,029,160 size\_test.h5
09/19/2013 04:16 PM 33,812 size\_test\_compressed.h5
>>> import gzip
>>> f = gzip.open('size\_test.csvz','w')
>>> df.to\_csv(f)
>>> f.close()
>>> ls size\_test.csvz # Ignore
09/19/2013 04:18 PM 10,533 size\_test.csvz
>>> from pandas import read\_csv
>>> df\_from\_csvz = read\_csv('size\_test.csvz',compression='gzip')
```

The final block of lines shows how a csv with gzip compression is written and directly read using pandas. This method also achieves a very high level of compression. Any NumPy array is easily written to a file using a single, simple line using pandas.

```
>>> x = randn(100,100)
>>> DataFrame(x).to\_csv('numpy\_array.csv',header=False,index=False)
```

1.3.3 HDFStore

`HDFStore` is the Class that underlies `to_hdf`, and is useful for storing multiple Series or DataFrames to a single HDF file. It's use is similar to that of any generic file writing function in Python – it must be opened, data can be read or written, and then it must be closed. Storing data is as simple as inserting objects into a dictionary. The basic use of a HDFStore for saving data is

```
>>> from pandas import HDFStore
>>> store = HDFStore('store.h5',mode='w',complib='zlib',complevel=6)
```

which opens the HDFStore named store for writing (mode='w') with compression. Stores can also be opened for reading (mode='r') or appending (mode='a'). When opened for reading, the compression options are not needed. Data can then be stored in the HDFStore using dictionary syntax.

```
>>> store['a'] = DataFrame([[1,2],[3,4]])
>>> store['b'] = DataFrame(np.ones((10,10)))
and finally the store must be closed.
>>> store.close()
```

The data can then be read using similar commands,

```
>>> store = HDFStore('store.h5',mode='r')
>>> a = store['a']
>>> b = store['b']
>>> store.close()
```

which will read the data with key 'a' in a variable named a, and similarly for b. A slightly better method for using a store is to use the Python keyword with and `get_store`. This is similar to opening the store, reading/writing some data, and then calling close(), only that the close() is automatically called when the store is no longer required. For example, with pd.get_store('store.h5') as store:

```
a = store['a']
b = store['b']
```

is equivalent to the previous code block, only the `close()` is called implicitly after the variables are read. `get_store` can be used with the same keyword arguments as HDFStore to enable compression or set the mode for opening the file.

1.4 Graphics

pandas provides a set of useful plotting routines based on matplotlib which makes use of the structure of a DataFrame. Everything in *pandas* plot library is reproducible using matplotlib, although often at the cost of additional typing and code complexity (for example, axis labeling).

plot

`plot` is the main plotting method, and by default will produce a line graph of the data in a DataFrame. Calling `plot` on a DataFrame will plot all series using different colours and generate a legend. A number of keyword argument are available to affect the contents and appearance of the plot.

- `style`, a list of matplotlib styles, one for each series plotted. A dictionary using column names as keys and the line styles as values allows for further customization.
- `title`, a string containing the figure title.
- subplots, a Boolean indicating whether to plot using one subplot per series (True). The default it False.
- legend, a Boolean indicating whether to show a legend
- `secondary_y`, a Boolean indicating whether to plot a series on a secondary set of axis values. See the example below.
- `ax`, a matplotlib axis object to use for the plot. If no axis is provided, then a new axis is created.
- kind, a string, one of:
  - '`line`', the default
  - '`bar`' to produce a bar chart. Can also use the keyword argument stacked=True to produce a stacked bar chart.
  - '`barh`' to produce a horizontal bar chart. Also support stacked=True.
  - '`kde`' or '`density`' to produce a kernel density plot.

1.4.1 `hist`

`hist` produces a histogram plot, and is similar to producing a bar plot using the output of `value_count`.

1.4.2 boxplot

`boxplot` produces box plots of the series in a DataFrame.

1.4.3 scatter_plot

`scatter_plot` produce a scatter plot from two series in a DataFrame. Three inputs are required: the DataFrame, the column name for the x-axis data and the column name for the y-axis data. `scatter_plot` is located in `pandas.tools.plotting`.

1.4.4 scatter_matrix

`scatter_matrix` produces a *n by n* set of subplots where each subplot contains the bivariate scatter of two series. One input is required, the DataFrame. `scatter_matrix` is located in `pandas.tools.plotting`. By default, the diagonal elements are histograms, and the keyword argument `diagonal='kde'` produces a ***kernel density plot***.

1.4.5 lag_plot

`lag_plot` produces a scatter plot of a series against its lagged value. The keyword argument `lag` chooses the lag used in the plot (default is 1). This is quite useful in ***Time Series Analysis***.