Data Wrangling with Python and Pandas

January 25, 2015

1 Introduction to Pandas: the Python Data Analysis library

This is a short introduction to pandas, geared mainly for new users and adapted heavily from the "10 Minutes to Pandas" tutorial from http://pandas.pydata.org. You can see more complex recipes in the Pandas Cookbook: http://pandas.pydata.org/pandas-docs/stable/cookbook.html

1.1 Initial Setup

Let's start by importing 3 useful libraries: pandas, numpy, and matplotlib

```
>>> [] import pandas
```

```
>>> [] import numpy
```

```
>>> [] import matplotlib.pyplot
```

1.2 Object Creation

You can create a `Series` by passing a list of values. By default, pandas will create an integer index.

```
>>> [] s = pandas.Series([1,3,5,numpy.nan,6,8])
```

```
>>> [] s
```

You can also use pandas to create an series of `datetime` objects. Let's make one for the week beginning January 25th, 2015:

```
>>> [] dates = pandas.date_range('20150125',periods=7)
```

```
>>> [] dates
```

Now we'll create a `DataFrame` using the `dates` array as our index, fill it with some random values using numpy, and give the columns some labels.

```
>>> [] df = pandas.DataFrame(numpy.random.randn(7,4), index=dates, columns={'dog','cat','mouse','duck'}
```

```
>>> [] df
```

It can also be useful to know how to create a `DataFrame` from a `dict` of objects. This comes in particularly handy when working with JSON-like structures.

```
>>> [] df2 = pandas.DataFrame({ 'A' : 1.,
                               'B' : pandas.Timestamp('20130102'),
                               'C' : pandas.Series(1,index=list(range(4)),dtype='float32'),
                               'D' : numpy.array([3] * 4,dtype='int32'),
                               'E' : pandas.Categorical(["test","train","test","train"]),
                               'F' : 'foo' })
```

```
>>> [] df2
```

1.3 Exploring the data in a DataFrame

We can access the data types of each column in a DataFrame as follows:

```
>>> [] df2.dtypes
```

We can display the index, columns, and the underlying numpy data separately:

```
>>> [] df.index
```

```
>>> [] df.columns
```

```
>>> [] df.values
```

To get a quick statistical summary of your data, use the .describe() function:

```
>>> [] df.describe()
```

1.4 Some basic data transformations

DataFrames have a built-in transpose:

```
>>> [] df.T
```

We can also sort a DataFrame along a given data dimension. For example, we might want to sort by the values in column B:

```
>>> [] df.sort(columns='dog')
```

We can also sort the rows (axis = 0) and columns (axis = 1) by their index/header values:

```
>>> [] df.sort_index(axis=0, ascending=False)
```

```
>>> [] df.sort_index(axis=1)
```

1.5 Selection

To select only only the first few rows of a DataFrame, use the .head() function

```
>>> [] df.head()
```

To view only the last few rows, use the .tail() function. Note that by default, both .head() and .tail() return 5 rows. You can also specify the number you want by passing in an integer.

```
>>> [] df.tail(2)
```

Selecting a single column yields a Series, equivalent to df.cat:

```
>>> [] df['cat']
```

We can also select a subset of the rows using slicing. You can select either by integer indexing:

```
>>> [] df[1:3]
```

Or by value (for example, slicing on a date range might come in handy):

```
>>> [] df['20150127':'20150129']
```

To select more than one column at a time, try .loc[]):

```
>>> [] df.loc[:,['cat','dog']]
```

And of course, you might want to do both at the same time:

```
>>> [] df.loc['20150127':'20150129',['cat','dog']]
```

1.6 Boolean Indexing

Sometimes it's useful to be able to select all rows that meet some criteria. For example, we might want all rows where the value of `cat` is greater than 0:

```
>>> [] df[df['cat'] > 0]
```

Or perhaps we'd like to eliminate all negative values:

```
>>> [] nonneg_only = df[df > 0]
```

```
>>> [] nonneg_only
```

And then maybe we'd like to drop all the rows with missing values:

```
>>> [] nonneg_only.dropna()
```

Oops... maybe not. How about we set them to zero instead?

```
>>> [] nonneg_only.fillna(value=0)
```

But what if your values aren't numeric? No problem, we can also do filtering. First, let's copy the `DataFrame` and add a new column of nominal values:

```
>>> [] df2 = df.copy()
```

```
>>> [] df2['color']=['blue', 'green','red','blue','green','red','blue']
```

```
>>> [] df2
```

Now we can use the `.isin`() function to select only the rows with 'green' or 'blue' in the `color` column:

```
>>> [] df2[df2['color'].isin(['green','blue'])]
```

1.7 Basic Math

It's simple to get the mean across all numeric columns:

```
>>> [] df.mean()
```

We can also perform the same operation on rows:

```
>>> [] df.mean(1)
```

Median also behaves as expected:

```
>>> [] df.median()
```

You can also use the `.apply`() function to evaluate functions to the data. For example, we might want to perform a cumulative summation (thanks, numpy!):

```
>>> [] df.apply(numpy.cumsum)
```

Or apply your own function, such as finding the spread (max value - min value):

```
>>> [] df.apply(lambda x: x.max() - x.min())
```

1.8 Combining DataFrames

Combining DataFrame objects can be done using simple concatenation (provided they have the same columns):

```
>>> [] frame_one = pandas.DataFrame(numpy.random.randn(5, 4))
```

```
>>> [] frame_one
```

```
>>> [] frame_two = pandas.DataFrame(numpy.random.randn(5, 4))
```

```
>>> [] frame_two
```

```
>>> [] pandas.concat([frame_one, frame_two])
```

If your DataFrames do not have an identical structure, but do share a common key, you can also perform a SQL-style join using the .merge() function:

```
>>> [] left = pandas.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
```

```
>>> [] left
```

```
>>> [] right = pandas.DataFrame({'key': ['foo', 'foo', 'bar'], 'rval': [3, 4, 5]})
```

```
>>> [] right
```

```
>>> [] pandas.merge(left, right, on='key')
```

1.9 Grouping

Sometimes when working with multivariate data, it's helpful to be able to condense the data along a certain dimension in order to perform a calculation for efficiently. Let's start by creating a somewhat messy DataFrame:

```
>>> [] foo_bar = pandas.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar','foo', 'bar', 'foo', 'foo'],
                                   'B' : ['one', 'one', 'two', 'three', 'two', 'two', 'one', 'three'],
                                   'C' : numpy.random.randn(8),
                                   'D' : numpy.random.randn(8)})
```

```
>>> [] foo_bar
```

Now let's group by column A, and sum along the other columns:

```
>>> [] foo_bar.groupby('A').sum()
```

Note that column B was dropped, because the summation operator doesn't make sense on strings. However, if we wanted to retain that information, we could perform the same operation using a hierarchical index:

```
>>> [] grouped = foo_bar.groupby(['A','B']).sum()
```

```
>>> [] grouped
```

The .stack() function can be used to "compress" a level in the DataFrame's columns:

```
>>> [] stacked = grouped.stack()
```

```
>>> [] stacked
```

To uncompress the last column of a stacked DataFrame, simply call .unstack():

```
>>> [] stacked.unstack()
```

1.10 Time Series

pandas has simple, powerful, and efficient functionality for performing resampling operations during frequency conversion (for example, converting secondly data into minutely data). Firse, let's create an array of 150 `dateTime` objects at a frequency of 1 second:

```
>>> [] rng = pandas.date_range('1/1/2015', periods=150, freq='S')
```

```
>>> [] rng
```

Now we'll use that to greate a time series, assigning a random integer to each element of the range:

```
>>> [] time_series = pandas.Series(numpy.random.randint(0, 500, len(rng)), index=rng)
```

```
>>> [] time_series.head()
```

Next, we'll resample the data by binning the one-second raws into minutes (and summing the associated values):

```
>>> [] time_series.resample('1Min', how='sum')
```

We also have support for time zon conversion. For example, if we assume the original time_series was in UTC:

```
>>> [] ts_utc = time_series.tz_localize('UTC')
```

```
>>> [] ts_utc.head()
```

We can easily convert it to Eastern time:

```
>>> [] ts_utc.tz_convert('US/Eastern').head()
```

1.11 Reading/Writing to files

Writing to a file is straightforward:

```
>>> [] ts_utc.to_csv('foo.csv')
```

As is reading:

```
>>> [] new_frame = pandas.read_csv('foo.csv')
```

```
>>> [] new_frame.head()
```

We can also read/write to Excel:

```
>>> [] new_frame.to_excel('foo.xlsx', sheet_name='Sheet1')
```

```
>>> [] pandas.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_values=['NA']).head()
```

But what if the data is a little... messy? Something like this:

```
>>> [] broken_df = pandas.read_csv('datasets/bikes.csv')
```

```
>>> [] broken_df[:3]
```

No problem! The $\texttt{read}_{\texttt{c}}\texttt{sv}$() function has lots of tools to help wrangle this mess. Here we'll

```
- change the column separator to a ;
- Set the encoding to 'latin1' (the default is 'utf8')
- Parse the dates in the 'Date' column
- Tell it that our dates have the date first instead of the month first
- Set the index to be the 'Date' column
```

```
>>> [] fixed_df = pandas.read_csv('datasets/bikes.csv', sep=';', encoding='latin1',
parse_dates=['Date'], dayfirst=True, index_col='Date')
```

```
>>> [] fixed_df.head()
```

1.12 Scraping data from the web

Many of you will probably be interested in scraping data from the web for your projects. For example, what if we were interested in working with some historical Canadian weather data? Well, we can get that from: http://climate.weather.gc.ca using their API. Requests are going to be formatted like this:

```
>>> [] url_template = "http://climate.weather.gc.ca/climateData/bulkdata_e.html?format=csv&stationID=54
```

Note that we've requested the data be returned as a `csv`, and that we're going to supply the month and year as inputs when we fire off the query. To get the data for March 2013, we need to format it with month=3, year=2012:

```
>>> [] url = url_template.format(month=3, year=2012)
```

This is great! We can just use the same read_csv function as before, and just give it a URL as a filename. Awesome.

Upon inspection, we find out that there are 16 rows of metadata at the top of this CSV, but pandas knows CSVs are weird, so there's a skiprows options. We parse the dates again, and set 'Date/Time' to be the index column. Here's the resulting dataframe.

```
>>> [] weather_mar2012 = pandas.read_csv(url, skiprows=16, index_col='Date/Time',
parse_dates=True, encoding='latin1')
```

```
>>> [] weather_mar2012.head()
```

As before, we can get rid of any comlumns that don't contain real data using .`dropna`()

```
>>> [] weather_mar2012 = weather_mar2012.dropna(axis=1, how='any')
```

```
>>> [] weather_mar2012.head()
```

Getting better! The Year/Month/Day/Time columns are redundant, though, and the Data Quality column doesn't look too useful. Let's get rid of those.

```
>>> [] weather_mar2012 = weather_mar2012.drop(['Year', 'Month', 'Day', 'Time', 'Data Quality'], axis=1)
       weather_mar2012[:5]
```

Great! Now let's figure out how to download the whole year? It would be nice if we could just send that as a single request, but like many APIs this one is limited to prevent people from hogging bandwidth. No problem: we can write a function!

```
>>> [] def download_weather_month(year, month):
           url = url_template.format(year=year, month=month)
           weather_data = pandas.read_csv(url, skiprows=16, index_col='Date/Time', parse_dates=True)
           weather_data = weather_data.dropna(axis=1)
           weather_data.columns = [col.replace('\xb0', '') for col in weather_data.columns]
           weather_data = weather_data.drop(['Year', 'Day', 'Month', 'Time', 'Data Quality'], axis=1)
           return weather_data
```

Now to test that this function does the right thing:

```
>>> [] download_weather_month(2012, 1).head()
```

Woohoo! Now we can iteratively request all the months using a single line. This will take a little while to run.

```
>>> [] data_by_month = [download_weather_month(2012, i) for i in range(1, 12)]
```

Once that's done, it's easy to concatenate all the dataframes together into one big dataframe using `pandas.concat()`. And now we have the whole year's data!

```
>>> [] weather_2012 = pandas.concat(data_by_month)
```

This thing is long, so instead of printing out the whole thing, I'm just going to print a quick summary of the `DataFrame` by calling `.info()`:

```
>>> [] weather_2012.info()
```

And a quick reminder, if we wanted to save that data to a file:

```
>>> [] weather_2012.to_csv('datasets/weather_2012.csv')
```

1.13 And there you have it! You're ready to wrangle some data of your own.