Data Processing with Pandas

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

In most of my past work, I've always had a need to solve various sorts of data analysis problems. Prior to discovering Python, AWK and other assorted UNIX commands were my tools of choice. These days, I'll mostly just code up a simple Python script (e.g., see the June 2012 *;login:* article on using the collections module). Lately though, I've been watching the growth of the Pandas library with considerable interest.

Pandas, the Python Data Analysis Library, is the amazing brainchild of Wes McKinney (who is also the author of O'Reilly's *Python for Data Analysis*). In short, Pandas might just change the way you work with data. Introducing all of Pandas in a short article is impossible here, but I thought I would give a few examples to motivate why you might want to look at it.

Preliminaries

To start using Pandas, you first need to make sure you've installed NumPy (http://numpy.scipy.org). If you've primarily been using Python for systems programming tasks, you may not have encountered NumPy; however, it gives Python a useful array object that serves as the cornerstone for most of Python's science and engineering modules (including Pandas). Unlike lists, arrays can only consist of a homogeneous type (integers, floats, etc.). Operations involving arrays also tend to operate on all of the elements at once. Here is a short example that illustrates some differences between lists and arrays:

```
>>> # Python lists
>>> c = [1,2,3,4]
>>> c * 3
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> c + [10,11,12,13]
[1, 2, 3, 4, 10, 11, 12, 13]
>>> import math
>>> [math.sqrt(x) for x in c]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
>>>

>>> # numpy arrays
>>> import numpy
>>> d = numpy.array([1,2,3,4])
>>> d * 3
```

```
array([ 3,  6,  9, 12])
>>> d + numpy.array([10,11,12,13])
array([11, 13, 15, 17])
>>> numpy.sqrt(d)
array([ 1.        ,  1.41421356,  1.73205081,  2.        ])
>>>
```

Once you've verified that you have NumPy installed, go to the Pandas Web site (http://pandas.pydata.org) to get the code before trying the examples that follow.

Analyzing CSV Data

One of my favorite pastimes these days is to play around with public data sets. Another one of my favorite activities has been riding around on my road bike—something that was recently curtailed after I hit a huge pothole and had to have my local bike shop build a new wheel. So, in the spirit of huge potholes, let's download the city of Chicago's pothole database from the data portal at http://data.cityofchicago.org. We'll save it to a local CSV file so that we can play around with it.

```
>>> u = urllib.urlopen("https://data.cityofchicago.org/api/views/7as2-ds3y/
rows.csv")
>>> data = u.read()
>>> len(data)
27683443
>>> f = open('potholes.csv','w')
>>> f.write(data)
>>> f.close()
>>>
```

As you can see, we now have about 27 MB of pothole data. Here's a sample of what the file looks like:

```
>>> f = open('potholes.csv')
>>> next(f)
'CREATION DATE,STATUS,COMPLETION DATE,SERVICE REQUEST NUMBER,TYPE OF
SERVICE REQUEST,CURRENT ACTIVITY,MOST RECENT ACTION,NUMBER OF POTHOLES
FILLED ON BLOCK,STREET ADDRESS,ZIP,X COORDINATE,
Y COORDINATE,Ward,Police District,Community Area,LATITUDE,LONGITUDE,LOCATI
ON\n'
>>> next(f)
'09/20/2012,Completed - Dup,09/20/2012,12-01644985,Pot Hole in Street,,,0,
172 W COURT PL,60602,1174935.20259427,1901041.84281984,42,1,32,
41.883847164125186,-87.63307578849374,"(41.883847164125186,
-87.63307578849374)"\n'
>>>
```

Pandas makes it extremely easy to read CSV files. Let's use its read_csv() function to grab the data:

```
>>> import pandas
>>> potholes = pandas.read_csv('potholes.csv', skip_footer=True)
>>> potholes
<class 'pandas.core.frame.DataFrame'>
Int64Index: 116718 entries, 0 to 116717
Data columns:
```

```
CREATION DATE                       116718  non-null values
STATUS                              116718  non-null values
COMPLETION DATE                     115897  non-null values
SERVICE REQUEST NUMBER              116718  non-null values
TYPE OF SERVICE REQUEST             116718  non-null values
CURRENT ACTIVITY                    94429   non-null values
MOST RECENT ACTION                  94191   non-null values
NUMBER OF POTHOLES FILLED ON BLOCK  93790   non-null values
STREET ADDRESS                      116717  non-null values
ZIP                                 115705  non-null values
X COORDINATE                        116660  non-null values
Y COORDINATE                        116660  non-null values
Ward                                116695  non-null values
Police District                     116695  non-null values
Community Area                      116696  non-null values
LATITUDE                            116660  non-null values
LONGITUDE                           116660  non-null values
LOCATION                            116660  non-null values
dtypes: float64(9), object(9)
>>>
```

When reading data, Pandas creates what's known as a DataFrame object. One way to view a DataFrame is as a collection of columns. In fact, you can easily extract specific columns or change the data:

```
>>> addresses = potholes['STREET ADDRESS']
>>> addresses[0:5]
0   172 W COURT PL
1   1413 W 17TH ST
2   11800 S VINCENNES AVE
3   3499 S KEDZIE AVE
4   1930 W CULLERTON ST
Name: STREET ADDRESS
>>> addresses[1] = '5412 N CLARK ST'
>>>
```

And there is so much more that you can do. For example, if you wanted to find the five most reported addresses for potholes, you could use this one-line statement:

```
>>> potholes['STREET ADDRESS'].value_counts()[:5]
4700 S LAKE PARK AVE     108
1600 N ELSTON AVE        84
7100 S PULASKI RD        80
1000 N LAKE SHORE DR     80
8300 S VINCENNES AVE     73
>>>
```

Let's say you want to find all of the unique values for a column. Here's how you do that:

```
>>> # Get possible values for the 'STATUS' field
>>> potholes['STATUS'].unique()
array([Completed - Dup, Completed, Open - Dup, Open], dtype=object)
>>>
```

Here is an example of filtering the data based on values for one of the columns:

```
>>> fixed = potholes[potholes['STATUS'] == 'Completed']
>>> fixed
<class 'pandas.core.frame.DataFrame'>
Int64Index: 94490 entries, 1 to 116717
Data columns:
CREATION DATE       94490  non-null values
STATUS              94490  non-null values
...
>>>
```

In this example, the relation `potholes['STATUS'] == 'Completed'` is computed across all 116,000 records at once and creates an array of Booleans. By using that array as an index into potholes, we get only those records that matched as True. It's kind of a neat trick.

In addition to street addresses, the pothole data also includes the total number of potholes fixed at each address. Let's try to refine our analysis so that it takes this into account. Specifically, we'd like to sum up the total number of potholes fixed at each address and base our report on that. Here's how to do it.

First, let's just pick out data on street addresses and number of potholes:

```
>>> addr_and_holes = fixed[['STREET ADDRESS',
...                  'NUMBER OF POTHOLES FILLED ON BLOCK']]
>>> addr_and_holes
<class 'pandas.core.frame.DataFrame'>
Int64Index: 94490 entries, 1 to 116717
Data columns:
STREET ADDRESS                         94489  non-null values
NUMBER OF POTHOLES FILLED ON BLOCK     93558  non-null values
dtypes: float64(1), object(1)
>>>
```

Next, let's drop missing values in the data:

```
>>> addr_and_holes = addr_and_holes.dropna()
>>> addr_and_holes
<class 'pandas.core.frame.DataFrame'>
Int64Index: 93558 entries, 13 to 116717
Data columns:
STREET ADDRESS                         93558  non-null values
NUMBER OF POTHOLES FILLED ON BLOCK     93558  non-null values
dtypes: float64(1), object(1)
>>>
```

Let's group the data by street address and calculate totals:

```
>>> addr_and_totals = addr_and_holes.groupby('STREET ADDRESS').sum()
>>> addr_and_totals[:5]
             NUMBER OF POTHOLES FILLED ON BLOCK
STREET ADDRESS
1 E 100TH PL        9
1 E 110TH PL       20
1 E 111TH ST       10
```

```
1 E 11TH ST           20
1 E 121ST ST          21
>>>
```

Finally, let's sort the results:

```
>>> addr_and_totals = addr_and_totals.sort('NUMBER OF POTHOLES FILLED ON
BLOCK')
>>> addr_and_totals[-5:]
                  NUMBER OF POTHOLES FILLED ON BLOCK
STREET ADDRESS
6300 N RAVENSWOOD AVE     461
8200 S MARYLAND AVE       498
3900 S ASHLAND AVE        575
12900 S AVENUE O          577
5600 S WOOD ST            664
>>>
```

And there you have it—the five worst blocks on which to ride your road bike. It's left as an exercise to the reader to take this data and extend it to find the worst overall street on which to ride your bike (by my calculation it's Ashland Avenue, which is probably of no surprise to Chicago residents).

A File System Example

Let's try an example involving a file system. Define the following function that collects information about files into a list of dictionaries:

```
import os

def summarize_files(topdir):
   filedata = []
   for path, dirs, files in os.walk(topdir):
      for name in files:
         fullname = os.path.join(path,name)
         if os.path.exists(fullname):
            data = {
               'path' : path,
               'filename' : name,
               'size' : os.path.getsize(fullname),
               'ext' : os.path.splitext(name)[1],
               'mtime' : os.path.getmtime(fullname)
               }
            filedata.append(data)
   return filedata
```

Now, let's hook it up to Pandas and use it to analyze the Python source tree:

```
>>> import pandas
>>> filedata = pandas.DataFrame(summarize_files("Python-3.3.0rc1"))
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4207 entries, 0 to 4206
Data columns:
ext         4207  non-null values
```

```
filename    4207  non-null values
mtime       4207  non-null values
path        4207  non-null values
size        4207  non-null values
dtypes: float64(1), int64(1), object(3)
```

Let's see how many files of different types there are:

```
>>> filedata['ext'].value_counts()[:5]
.py    1618
.c     479
.rst   429
.h     263
.o     236
>>>
```

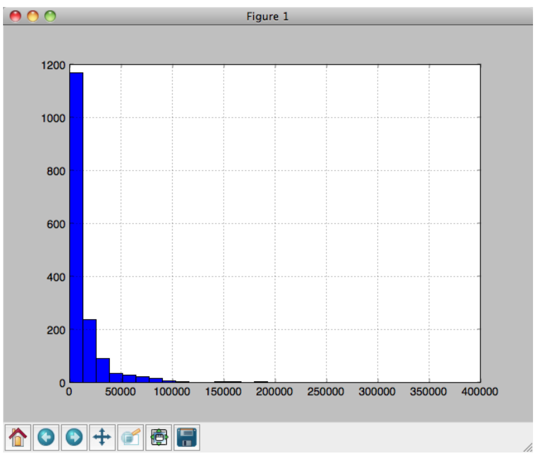

Figure 1: Histogram created with pyfiles['size'].hist(bins=30) using the matplotlib library

As a final example, let's generate a few statistics and use maplotlib to make a histogram. Here, we'll look at the sizes of .py files:

```
>>> pyfiles = filedata[filedata['ext'] == '.py']
>>> pyfiles['size'].max()
385802
>>> pyfiles['size'].mean()
12964.483930778739
>>> pyfiles['size'].std()
23799.089183395961
>>> pyfiles['size'].hist(bins=30)
<matplotlib.axes.AxesSubplot object at 0x102f0e290>
>>> import pylab
>>> pylab.show()
```

If it works, you'll end up with a plot that looks like Figure 1.

That's pretty neat—and it didn't involve much code.

Final Words and In Memoriam

If you're faced with the task of analyzing data, Pandas is definitely worth a look. Although all of the problems shown in this example could have been solved by short Python scripts, Pandas makes it even easier and more succinct.

Finally, in the last example, matplotlib (http://matplotlib.sourceforge.net) was used to make a plot. matplotlib is one of the most popular extensions to Python that is in widespread use by scientists and engineers. Sadly, John Hunter, the creator of matplotlib, passed away suddenly this past August from complications of cancer treatment, leaving behind his wife and three daughters. If you've benefited from the use of matplotlib, a memorial fund has been established. More information can be found at http://numfocus.org/johnhunter/.