

Data Analysis with Python

Jared Garst

November 29, 2015

Python is a high level scripting language. While there are many reasons to like or dislike Python, we are introducing it to you because it is becoming the *lingua franca* of science. This means that increasingly you can expect other people to understand your code, and for many of your problems to already be solved and documented in Python. Briefly, some exceptional Python features are:

Beautiful Syntax

Python will make your life easier, by encouraging simple, clean, easy to read code.

The Python Shell

If you don't need to keep your code, you can pass it to Python as a series of commands. This allows programmers to gracefully move their files and explore their data.

No Types

There is no mechanism to guarantee the data you store will retain important properties.

Interpreted

The code you write is the code that runs. Nothing will be checked, nothing will be optimised.

Program Control Through Indentation

Other languages use curly brackets {} to determine program flow, and tabs by convention for readability. Python conflates the two.

IPython is an expansion of the Python shell, designed for a more exploratory approach to programming than your typical editor. It gives you a unique structure, and tools to view your output. It also provides you with many tools that will help you become a better programmer outside this class - we won't get into many of those, but you can use the standard *tutorial* to learn them.

There are two versions of Python that you will encounter in the wild, Python 2 and Python 3. Python 3 is a take-no-prisoners reimagining of the the language. The result is a slightly smoother language, that breaks everything down to the print statement in Python 2. Most libraries support both Python 2 and 3, but most code is in Python 2. This means the appropriate version will depend on how old your project is. If you are unsure, use Python 2, and consider that unless you use lots

of `unicode`, switching to Python 3 will provide only a small benefit to your quality of life.

Setup and Startup

If you have never set up a programming environment before, the easiest way to start is to install `Anaconda`, distributed by Continuum Analytics. This modified python environment will install the most common python libraries. It is distributed for free in hopes of later selling packages that speed up your python code, but you can simply use it as a convenient installation tool. If you want to try a grittier, more controlled installation of Python and its many friends, you can schedule some time with `Jared`.

Once you have completed the installation, run `ipython notebook` in the terminal, or the Anaconda IPython Notebook executable. A web browser will automatically start with your notebook. Managing directories in notebooks is a little clumsy. Either make sure to run IPython in the directory you want, or `change the default directory` to match your workflow.

There are a few flavors of help commands available in IPython. Familiarity with the options allows you to reference the manuals, or explore specific parts of the language.

`help()` is the native python help command. Type `help()` into the first cell, and execute with shift-enter. This is a good place to spend time getting familiar with the language, but you can also ask it about specific objects. For example, run `help(list)` to see some common list manipulations.

`?` is IPython's own help command, and it is often nicer to use than `help`. Try running `?` to see an introduction to IPython. Like `help` you can also ask more specific commands, run `import numpy; numpy?` to see an overview of the exceptionally useful `numpy` module. If you need all the gory details of an object, `numpy??` will give you the source code.

`<regex>?` allows you to use `?` to search through an object. Try running `numpy.*cos*?` to see every function in `numpy` related to `cos`.

`%quickref` is a 'magic function', unique to IPython, that you can use to see a verbatim list of all special IPython commands, including the magic functions.

`<tab>` will provide code completion. Type but do not run `numpy.arc`

```
A quick start:
help() - help(list)
? - import numpy; numpy? - numpy??
numpy.*cos*? - numpy.arc<tab> -
Ctrl-m h
```

followed by a <tab> to have IPython suggest a list of inverse trig functions.

`Control-m h` Provides a list of hotkeys for IPython notebooks. If you spend a lot of time These can really speed up your programming,

First Steps

```

In [1]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        import scipy.stats as st
        %matplotlib inline

In [2]: capData = pd.read_csv('DataFile.csv')
        print capData.head()
        print capData.tail()

                                         *Some Numbers*

In [4]: allData = capData.stack()
        binwidth = 0.2;
        cmin = capData.min();
        cmax = capData.max()
        bins = np.arange(cmin, cmax, binwidth)

In [5]: fig, ax = plt.subplots();
        ax.set_xlim([cmin, cmax]);
        ax.hist(list(allData), bins=bins)

In [6]: N = len(allData)
        print "N is", N
        mu = np.mean(allData)
        print "mu is", "{0:.2f}".format(mu)
        var = np.var(allData)
        print "variance is", "{0:.2f}".format(var)

                                         N is 1536
                                         mu is 94.50
                                         variance is 3.02

In [7]: area = N * binwidth
        params = {'loc':mu, 'scale':np.sqrt(var)}
        fit = area * st.norm(**params).pdf(bins)
        ax.plot(bins, fit, 'r--', linewidth=3);
        fig

```

libraries

numpy, scipy, matplotlib, and pandas will help you calculate, reference common functions, plot, and wrangle data. `%matplotlib inline` is a 'magic function', which tells IPython how to display plots.

data cleaning

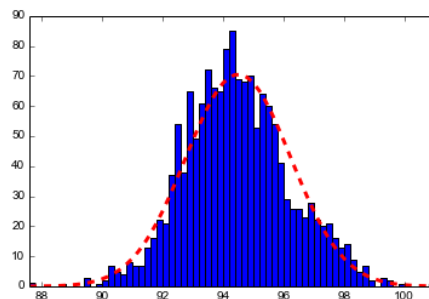
`head` and `tail` return the limits of the pandas dataFrame, and are being used to check that the csv file doesn't have unexpected cruft at the ends. Always check that you are importing your intended data.

format

adds external variables to strings, at specified precision. `{0:.2f}` is replaced by the first float argument of `format`, rounded to two decimal places.

argument unpacking

`function(*iter)` passes the elements of `itr` to function in order. `function(**dict)` passes them based on their keys.



Python Gotchas

Some of the ways that python makes your life easier can send your program off the rails, if you don't understand what the computer is doing.

TYPES still exist in python, even if you can't enforce them. Consider the following python code:

```
In [1]: print 3/2, 3.0/2.0
1 1.5
```

In the underlying C code, an operation on two integers always returns an integer, because floating point¹ math is much slower. In python, you can check a type with `type(object)`, and enforce a type with `assert` and `isinstance`.

```
In [2]: print type(3), type(3.0)
<type 'int'> <type 'float'>

In [3]: assert isinstance(3, int)
```

`assert` will stop the program with an error if the next statement is not true. Python 3 is smarter about division, and will return 1.5 for both calculations. In the likely event that you cannot use python 3, you can still import the behavior by including

```
from __future__ import division
```

at the top of your file or notebook.

COPYING OBJECTS can be done in two distinct ways, shown abstractly in Figure 1. The more common method is to copy the object reference, so that it can be used in a different part of the code. The second method is when you want to have two distinct copies, so that you can have both an original version and a modified version. In the first case it is appropriate to duplicate the pointer. In the second case it is necessary to duplicate the object itself.

Because the first case is common and the second case is expensive, python duplicates the reference by default.

```
In [1]: import numpy as np
In [2]: lst1 = np.array(['An Object'],
                        dtype=object);
        lstcpy = lst1; lst2 = lst1.copy()
In [3]: print lst1, lstcpy, lst2
['An Object'] ['An Object'] ['An Object']
In [4]: lstcpy[0] = 'A New Object'
In [5]: print lst1, lstcpy, lst2
['A New Object'] ['A New Object'] ['An Object']
```

¹ Float stands for 'floating point number', otherwise known as a decimal number. Though not usually relevant for data analysis, be aware that `int` and `float` have detailed differences in precision and speed.

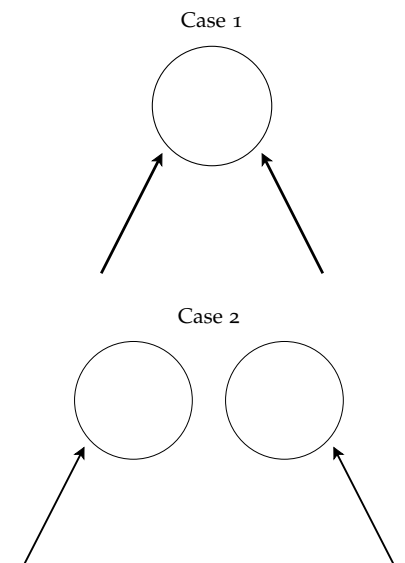


Figure 1: The finger pointing at the moon is not the moon

two types of variables
The copying code produces different results when the strings are not wrapped in numpy arrays. Why?

THE NOTEBOOK STATE is changed every time you run a command, not every time you add a cell. This allows you to create confusing code, by removing cells or executing them in a strange order.

```
In [3]: message = 'IPython does not have to run
        from top to bottom'
In [1]: message = 'Good luck debugging this'
In [4]: print message
        IPython does not have to run from top to bottom
In [2]: print message
        Good luck debugging this
```

You can debug this problem by paying attention to the numbers in front of the cells. Commands are executed in sequence with their numbers, not their position. Try to keep your notebook linear, by not changing cells that have already successfully executed.

Problems

1 DOCUMENTATION FROM PYTHON

Use the help commands to locate the `scipy.stats` documentation on the Poisson distribution. Is the distribution a function or an object? Plot² the expected number of counts for 10 trials of a poisson process, each with an average of 3 counts per trial.

² You will need the syntax for matplotlib. You can find everything with the help commands, or can get syntax and ideas from [example plots](#) that others have made.

2 DOCUMENTATION FROM GOOGLE

Try calling the poisson distribution function from `scipy.stats` with no arguments. You will get an error message saying one argument was passed, but two were needed. Use google to explain these strange parameter counts.

3 USING PYTHON TO CALCULATE

Reproduce the mean, variance, and standard deviation of the data using your own functions.

4 OBSERVE THE CENTRAL LIMIT THEOREM

Write a function `dataAvg(list, n)` that returns a new list \bar{x}_n , containing all possible averages of n values from `list`. Use this to plot \bar{x}_2 , \bar{x}_{10} and \bar{x}_{100} . What are the means of these averaged distributions? What are the variances? How do these compare to the mean and variance of x ?

More Reading

Data Structures and List Manipulation

Units and Errors

Source control: Mercurial and Git

Lambda Functions and Functional programming