

Introduction to Python for scientists

Experimental physics / Advanced lab series, CSU-East Bay Physics Department

Author: Ryan Smith

Python is a pretty cool language. It's used in industry and in the tech world, is free, and has libraries to do powerful scientific stuff with a few lines of code.

In this guide we'll walk through some of the basics of using python for scientific applications – specifically, we are thinking about plotting and analyzing data taken in an experimental setting.

Progression of concepts in learning python:

- opening python, doing some basic math
- print hello world
- scripts that compute stuff
- importing libraries
- defining an array (numpy)
- plotting cos function
- functions learned:
 - ➔ `np.size`, `np.arange`, `np.cos`, `np.sin`, `np.pi`, `np.genfromtxt`, `np.sqrt`
 - ➔ `plt.plot`, `plt.xlabel`, `plt.ylabel`, `plt.title`, `plt.grid`, `plt.savefig`, `plt.show()`
 - ➔ self-check: which functions are from numpy library, which from matplotlib?
- importing and plotting data
- plotting in multiple panes
- error analysis: quantifying uncertainty
- make a random array – `np.random.random((100,1))`
- mean: `np.mean`
- standard deviation: `np.std`
 - ➔ check your understanding: 68.2% of points in a normal distribution fall within +/- sigma.. can you explain this?
- central limit theorem – standard deviation and sigma revisited
- fitting -- linear fit, nonlinear fit
- χ^2 and its mission -- check if there is something wrong with the model or the data
 - ➔ https://en.wikipedia.org/wiki/Goodness_of_fit
 - ➔ s value
- propagation of errors -- how to do with python as an activity..

things that still need to be included/fleshed out: importing functions from another file; χ^2 .

*There are multiple IDEs such as python(x,y) or anaconda. Here's a link to download anaconda:

<https://www.continuum.io/downloads> (this can be for window, os x, or linux)

Install. then open the Navigator and open Spyder.

I think using python 2.7 for now is good.

I assume that you have already downloaded and installed a python interactive development environment (IDE) user interface. If not, see footnote below*.

Let's start by making sure that your python is working by typing in something into the command prompt. Type into the command prompt `print("Hello World!")` then hit enter.

It should look something like the following window:



[This image is from this [blog](#).]

Notice how it did *exactly* what you asked it to do - print some text to the screen. Also try typing (in the command prompt)

```
In [2]: a=5
```

then hit enter. (I'll just use "In [2]:" as a way to indicate typing into the command line from now on. I realize that number 2 increases with each new line...)

Then try

```
In [2]: a+a
```

and it should return 10. You are already doing some programming!

Writing scripts - plot something

Why scripts? Couldn't you just tell python to do stuff through the command prompt only? You may want to run many commands, and you may want to run a sequence of them and then change something in a sequence and try again. That's why scripts are a good idea. Here is a basic script to plot something - I am also making extra comments along the way. Green stuff in this doc is commented out (may look slightly different in your IDE) so this is invisible to python when you run a file. Notice how either `#` or `"""` will comment things out.

Good code will always be commented in the code itself. This lets you and other users know what your code is doing. Here is a typical heading - this should have information about what the file is, what it does, and often some example cases that tell you how to use the script.

```
1. """
2. filename: simple_plot_sine_wave.py
3. Created on Mon Apr 11 14:10:24 2016
4. @author: Ryan Smith
5.
6. This is a script to practice the fundamentals of plotting in python
7. using the numpy and matplotlib libraries.
8.
9. This is the simple plotting demonstration that we did in class.
10. """
```

(these scripts were formatted for Word with an online converter:

<http://www.planetb.ca/syntax-highlight-word>)

Since it may be your first time to run a script, let's not worry about all of the details just yet but try to run a few lines of code. First, you need to open a new script (if not already open) by choosing "New File" in the File menu.

We need to import some libraries for doing what we need to do. Try typing these lines in to your new script:

```
1. # import relevant packages
2. import numpy as np
3. import matplotlib.pyplot as plt
```

numpy means numerical python. **matplotlib** is a library that makes nice plots (like matlab, a commercial program that is also good, but is not free). Importing both gives us some helpful functionalities, such as doing some interesting math and making pretty plots of our results.

Next we will define an x-axis variable and then make a function of that variable. `np.arange(start,stop,step)` is a function in the numpy library that outputs an array starting from the `start` argument, and making steps of size `step` until reaching `stop` (not including this value). If you just clicked on the hyperlink for `np.arange` and saw that it is written as `numpy.arange` and are perplexed, don't worry. We imported everything from numpy as np so we just need to type "np." to denote that we are using a function from the `numpy` library. This may seem cumbersome at first, but it is actually very helpful as

```
1. # define arrays
2. time = np.arange(0.0, 5.0, 0.1)
3. voltage = np.cos(2*np.pi*time)
```

The same is also true for the cosine - it's not built in to python proper, but it is a part of numpy. So if we had just typed in `cos(pi)` into the command prompt first instead of `print("Hello World!")` then we would not have gotten the value -1.0 ... python would have tossed us an error because it had no idea what this cos function was that we were talking about (also pi - it's a part of numpy but regular python doesn't know what pi is). But simply by calling on the numpy library through `np.<some function>`, you can indicate that this is a function included in that library - try it! This avoids confusion when working with other libraries.

Does the following command work?

```
In [2]: cos(pi)
```

Nope. Error. How about this?

```
In [3]: np.cos(np.pi)
```

Yep. Note that this only works because you have already run the line

```
1. import numpy as np
```

If you open up python again or restart the kernel, you'll need to import the library again for this to work. This is why you'll see that we import libraries for every script that we create.

Now let's try running our script that we made so far (4 lines of code - two to load libraries, and two to make our variables). In windows (and some macs) you can hit F5. Or click on the green play button above to run. Hmm.. Nothing seems to happen. But something did happen. Go to the command prompt and type (and hit enter after)

```
In [2]: time
```

Something did happen! That's our array that we just created, and it's stored in the memory of python.

Try also

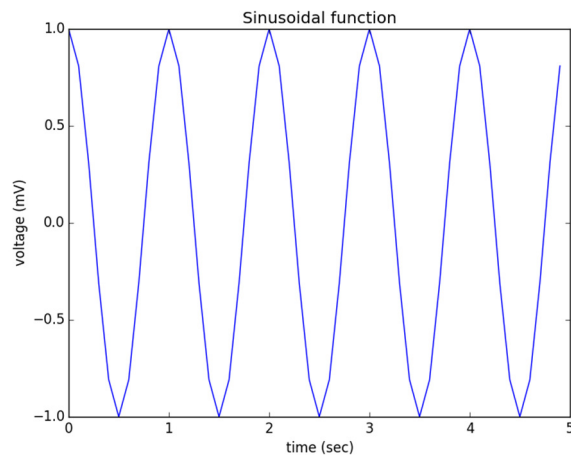
```
In [2]: np.size(time)
```

Now you know how big your **time** array is. Is **voltage** the same size? Check that!

Now it's time to plot our data - very simple:

```
11.      # plot our data
12.      plt.plot(time, voltage)
```

You may not immediately see your plot. But if you look in the list of windows (on Windows it's the taskbar below) you can click on it, and you should see something like this:



You may also see your plot inline in your command window. I prefer plotting in an outside window - this allows zooming in to see features and I just find it all-around more convenient. To make this happen, type (in the command prompt):

```
In [2]: %matplotlib qt
```

And if you want it to be back to plotting inline, just type:

```
In [2]: %matplotlib inline
```

Ok, let's get a little more nuanced. This is a good start - after loading the two libraries, just three lines of code got us a decent plot. Let's make it nicer. Instead of just `plot(time, voltage)` let's do the following:

```
21.      # plot our data
22.      plt.figure(1)
23.      plt.clf()
24.      plt.plot(time, voltage)
25.      plt.xlabel('time (sec)')
```

```
26. plt.ylabel('voltage (mV)')
27. plt.title('Sinusoidal function')
28. plt.grid(True)
29.
30. # save the figure as a png file. It will be saved
31. # to the same directory as your program.
32. plt.savefig("simple_plot.png")
33.
34. # show the plot
35. plt.show()
```

When you run this, you can see that there are some nice things happening here. This is starting to look like a nice graph, almost publication worthy.

😊

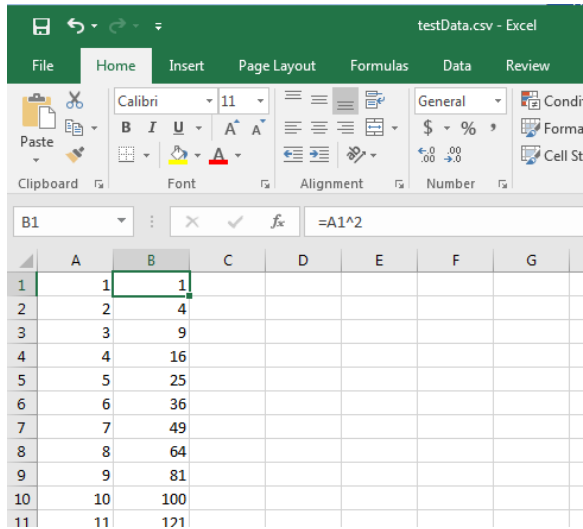
Here is the whole code to create and plot some data:

```
1. """
2. filename: 01 simple_plot_sine_wave.py
3. Created on Mon Apr 11 14:10:24 2016
4. @author: Ryan Smith
5.
6. This is a program to practice the fundamentals of plotting in python
7. using the matplotlib package.
8.
9. This is the simple plotting demonstration that we did in class.
10. """
11. # import libraries
12. import matplotlib.pyplot as plt
13. import numpy as np
14.
15. # define variables
16. time = np.arange(0.0, 5.0, 0.1)
17. voltage = np.cos(2*np.pi*time)
18.
19. # plot our data
20. plt.figure(1)
21. plt.clf()
22. plt.plot(time, voltage)
23. plt.xlabel('time (sec)')
24. plt.ylabel('voltage (mV)')
25. plt.title('Sinusoidal function')
26. #plt.grid(True)
27.
28. # save the figure as a png file. It will be saved
29. # to the same directory as your program.
30. plt.savefig("simple_plot.png")
31.
32. # show the plot
33. plt.show()
```

But this is not “real” data. We need to import some data from a file in order to do research –for example, finding the position of peaks in the data, rescaling, finding the area under a curve, fitting a model to the data, performing a fast Fourier Transform (FFT) to look at spectral content of a signal, histogramming data, etc. We will begin to approach how to do that next.

Importing and plotting data

Next we will try importing some data. You will need to make a *.csv file - comma-delimited values. If you have Excel (or similar), make a file that has two columns - it doesn't really matter what the columns are, but for our example, make the first column be 1,2,3,4,... and the second column be the square of the first column.



	A	B	C	D	E	F	G
1	1	1					
2	2	4					
3	3	9					
4	4	16					
5	5	25					
6	6	36					
7	7	49					
8	8	64					
9	9	81					
10	10	100					
11	11	121					

Then go to Save As... and save this as a *.csv (comma-delimited) file in the same directory as your python scripts.

Your data doesn't need to be made in Excel - it just has to look like this when opened with a text editor:

```
1,1
2,4
3,9
4,16
5,25
6,36
etc.
```

We will need to be sure that our python files (*.py) and files that we are working with initially are all in the same directory. It is possible to import data files from other directories too, but let's start simple first...

Here is the whole code that we will use to import our data and plot it:

```
1. """
2. filename: 02 import_and_plot_data.py
3. Created on Wed Apr 13 13:18:29 2016
4. @author: Ryan Smith
5.
6. This script will import some comma-delimited data,
7. the plot it and save the result as a *.png file
8. in the same directory.
9. """
10. # import libraries
11. import matplotlib.pyplot as plt
12. import numpy as np
13.
14. #####
15. # user-modified area:
16. filename= 'testData.csv'
17. xaxis_label = 'time (ms)'
18. yaxis_label = 'signal (uV)'
19. titleName = 'plot of my interesting data'
20. #####
21.
22. # import data and assign values to the columns
23. data = np.genfromtxt(filename,delimiter=',')
24. x_values = data[:,0]
25. y_values = data[:,1]
26.
27. y_values = np.sqrt(y_values)
28.
29. # plotting
30. plt.figure(1)
31. plt.clf()
32. plt.plot(x_values,y_values,'*')
33. plt.xlabel(xaxis_label,fontsize=15)
34. plt.ylabel(yaxis_label,fontsize=15)
35. plt.grid()
36. plt.title(titleName,fontsize=20)
37.
38. plt.savefig('mydata.png')
```

We did a couple new things as well, like increase the font sizes and putting dots where the data points are. Hopefully you can see what does what by playing with the code and checking the results. We also took the square root of our data so that what was a quadratic now looks linear...

For reference:

If you want to look up any of the help files for plotting-related things, you can search `plt.<function name>` to find things in the `matplotlib.pyplot` library.

For example, search online (favorite search engine) for [plt.plot](#) .

All of the pyplot functions can be found here:

http://matplotlib.org/api/pyplot_summary.html%20

Importing and analyzing and plotting data

More code! This is a re-do *almost* of our previous, but with some changes. This time we “analyze” our data (modify it from its original form).

```
1. """
2. filename: 03 import_analysis_and_plot_data.py
3. Created on Wed Apr 13 13:18:29 2016
4.
5. @author: Ryan Smith
6. """
7. # import libraries
8. import numpy as np
9. import matplotlib.pyplot as plt
10.
11. #####
12. # user-modified area:
13. filename= 'testData.csv'
14. xaxis_label = 'time (ms)'
15. yaxis_label = 'signal (uV)'
16. title = 'plot of my interesting data'
17. #####
18.
19. # import data
20. data = np.genfromtxt(filename,delimiter=',')
21. x_values = data[:,0]
22. y_values = data[:,1]
23.
24. y_values = np.sqrt(y_values)
25.
26. # modify data so that a couple points are altered
27. y_values[3] = y_values[3]*1.1
28. y_values[6] = y_values[6]*0.8
29.
30. # fitting procedure
31. p = np.polyfit(x_values,y_values,1)
32. m = p[0]
33. b = p[1]
34. yfit = m*x_values + b
35.
36. # plotting
37. plt.figure(1)
38. plt.clf()
39. plt.plot(x_values,y_values,'.-')
40. plt.plot(x_values,yfit,'.-r')
41. plt.xlabel(xaxis_label,fontsize=15)
42. plt.ylabel(yaxis_label,fontsize=15)
43. plt.grid()
44. plt.title(title,fontsize=20)
45. # save the figure as a png file. It will be saved
46. # to the same directory as your program.
47. plt.savefig('mydata.png')
48. # show the plot
49. plt.show()
```

As you can see, we did a little ‘analysis’ this time – we took the square root of our data, which was a quadratic. Now the plot looks like a line, as we would expect for taking the square root... Then we fit a line

to these data, and plot our line as well. See if you can understand every line of the previous code. Changing the code and seeing what happens is a great way to learn!

Plotting data in multiple panes

Here's a slightly fancier example of plotting stuff - this time a couple of panes in one figure:

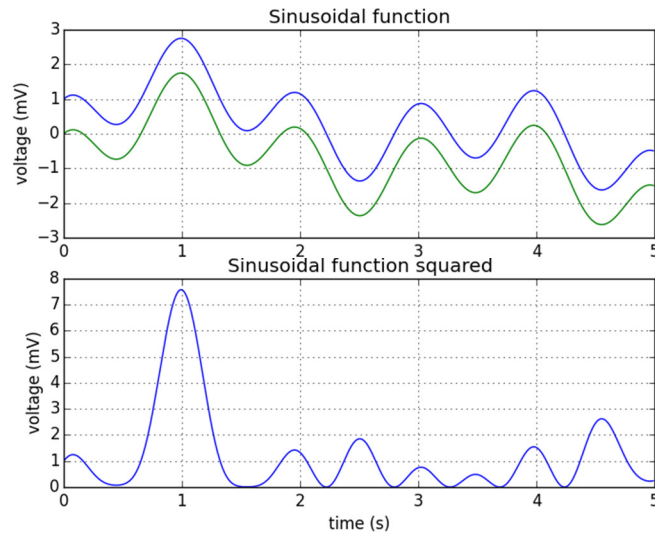
```
1. """
2. filename: 04_plot_two_window_panes.py
3. Created on Wed Apr 13 13:18:29 2016
4.
5. @author: Ryan Smith
6. """
7. # import libraries
8. import numpy as np
9. import matplotlib.pyplot as plt
10. # -*- coding: utf-8 -*-
11. """
12. Created on Mon Apr 11 14:10:24 2016
13.
14. @author: Ryan Smith
15.
16. This is a program to practice the fundamentals of plotting in python
17. using the matplotlib package.
18.
19. This one plots two different panes in the same figure.
20. """
21. # import libraries
22. import matplotlib.pyplot as plt
23. from numpy import *
24.
25. # define variables
26. t = np.arange(0.0, 5.0, 0.01)
27. s = np.cos(2*pi*t)+np.sin(t)+np.sin(2*t)
28.
29. # plot my data
30. plt.figure(1)
31. plt.clf()
32. plt.subplot(211)
33. plt.plot(t, s)
34. plt.plot(t, s-1)
35. plt.xlabel('time (s)')
36. plt.ylabel('voltage (mV)')
37. plt.title('Sinusoidal function')
38. plt.grid(True)
39.
40. # plot a second pane in the same figure -- this is s^2 (s**2 is the syntax)
41. plt.subplot(212)
42. plt.plot(t, s**2)
43. plt.xlabel('time (s)')
44. plt.ylabel('voltage (mV)')
45. plt.title('Sinusoidal function')
46.
47. # save the figure as a png file. It will be saved
48. # to the same directory as your program.
```

```

49. plt.savefig("test.png")
50.
51. # show the plot
52. plt.show()

```

The line subplot(212) -- the 21 means a 2 x 1 ('two by one,' 2 down, 1 across) arrangement of panes. The last 2 means focus on the 2nd one. This is one situation where we don't start our count with 0 in python.



output of the previous script

Error analysis: quantifying uncertainty

Refer to Bevington's [Data reduction and error analysis](#) or class notes on error analysis.

Quick review: the *mean* is the average of a bunch of numbers: add them up, divide by how many numbers. The *standard deviation* is the spread of these numbers, putting a value to how much the collection of numbers spread out from their mean. As we'll explore, if a collection of numbers follows a normal distribution (below), then there are some cool properties that help us to efficiently make decisions.

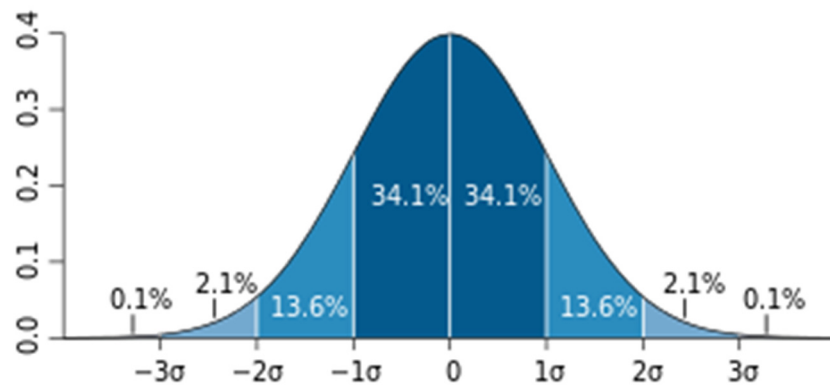


Figure 1. The "normal" distribution. Why this is so cool: If you have the mean and standard deviation of a collection of numbers that follow this distribution, then 68.2% of them will fall within $\pm \sigma$ of the mean.

Let's use our skills to analyze some data - let's start with making some random data, and checking the standard deviation and mean of this:

```

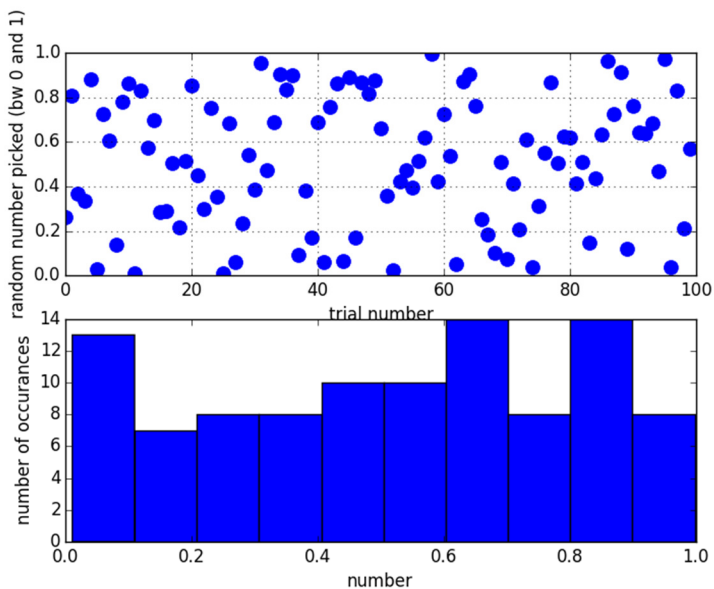
1. """
2. filename: 05 random numbers
3. Q: what are some are some statistical ways that we
4. can evaluate our random numbers?
5.
6. Created on Mon May 02 13:27:02 2016
7. @author: Ryan Smith
8. """
9. from __future__ import division
10. import numpy as np
11. import matplotlib.pyplot as plt
12.
13. #####
14. ## defining my data #####
15. #####
16. y_data = np.random.random((100,1))
17.
18.
19. #####
20. ## analysis of data #####
21. #####
22.
23. Npts = np.size(y_data)
24.
25. sigma = np.std(y_data)
26. meanval = np.mean(y_data)
27.
28. print "My standard dev is:",sigma
29. print "My average is:",meanval

```

```

30.
31. #####
32. ## plotting #####
33. #####
34.
35. plt.figure(1)
36. plt.clf()
37. plt.subplot(211)
38. plt.plot(y_data, '.', markersize=20)
39. plt.xlabel("trial number")
40. plt.ylabel("random number picked (bw 0 and 1)")
41. plt.grid(True)
42. plt.show()
43.
44. plt.subplot(212)
45. plt.hist(y_data, bins=Npts/10)
46. plt.xlabel("number")
47. plt.ylabel("number of occurrences")
48.
49. plt.savefig('random numbers.png')

```



My standard dev is: 0.287261733761

My average is: 0.510760773756

This code picks some random numbers, and then both plots and histograms them. It's reasonable that the average ended up being somewhere near 0.5, since we are picking random numbers between 0 and 1. And maybe you can imagine that the spread of numbers is around ± 0.3 . But do 68.2% of the points fall within \pm this sigma? Hmm.. I don't think so. This doesn't look like a normal distribution! Make it more numbers than 100 to convince yourself.

You could add this code at the end of your analysis section to count up the number of points between \pm sigma.

```

1. counter = 0
2. for k in range(0, Npts-1):
3.     if y_data[k] > meanval-sigma and y_data[k] < meanval+sigma:
4.         counter = counter + 1
5. print "Total number bw +/- sigma:",counter
6. print "Total fraction bw +/- sigma:",counter/Npts

```

You might find that no matter how many times you run your code you are going to get something lower than 68% of the points!

What if we instead took the mean value of a collection of random numbers, saved that value, and did that over and over? Wouldn't the average of these averages be hovering around 0.5? And perhaps this new collection of points follows a normal distribution!

```

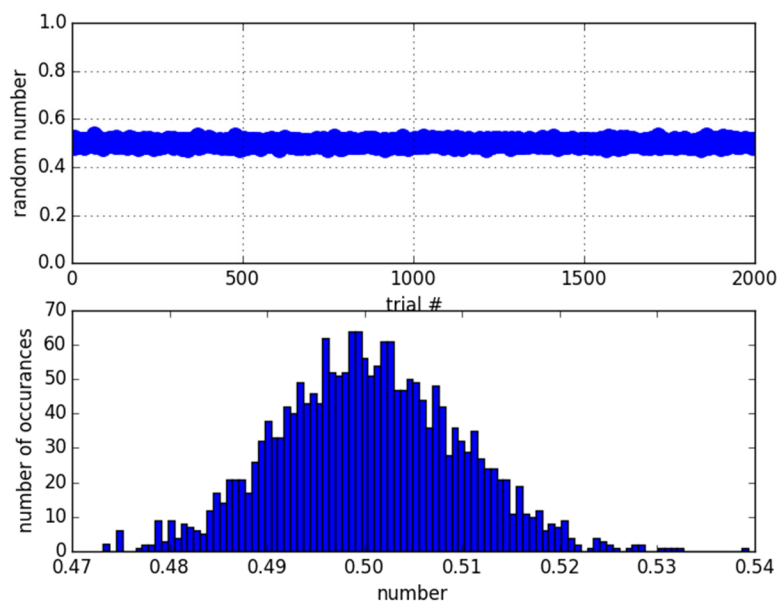
1. """
2. filename: 06 random numbers stats on mean
3. This time we take stats on the means of the data.
4. aka the central limit theorem!
5.
6. Created on Mon May 02 13:27:02 2016
7. @author: Ryan Smith
8. """
9. from __future__ import division
10. # import libraries
11. import numpy as np
12. import matplotlib.pyplot as plt
13.
14. meanvals = np.empty(0)
15. for j in range(0,2000):
16.     y_data = np.random.random((1000,1))
17.     meanvals = np.append(meanvals,mean(y_data))
18.
19. Npts = np.size(meanvals)
20.
21. sigma = np.std(meanvals)
22. meanval = np.mean(meanvals)
23.
24.
25. print "My standard dev is:",sigma
26. print "My average is:",meanval
27. counter = 0
28.
29. for k in range(0, Npts-1):
30.     if meanvals[k]> meanval-sigma and meanvals[k]< meanval+sigma:
31.         counter = counter + 1
32.
33. print "Total number bw +/- sigma:",counter
34. print "Total fraction bw +/- sigma:",counter/Npts
35.
36. plt.figure(1)
37. plt.clf()
38. plt.subplot(211)
39. plt.plot(meanvals, '.', markersize=20)
40. plt.grid(True)
41. plt.show()

```

```

42. plt.xlabel("trial #")
43. plt.ylabel("random number")
44. plt.ylim([0,1])
45.
46. plt.subplot(212)
47. plt.hist(meanvals, bins=100)
48. plt.xlabel("number")
49. plt.ylabel("number of occurrences")
50.
51. # save the figure as a png file. It will be saved
52. # to the same directory as your program.
53. plt.savefig("stats on mean.png")
54. # show the plot
55. plt.show()

```



My standard dev is: 0.00922160089573

My average is: 0.500065541658

Total number bw +/- sigma: 3405

Total fraction bw +/- sigma: 0.681

Oh! Interesting. It indeed does follow a normal distribution. This leads us to discuss the central limit theorem... Try playing around the number of bins (currently 100) and the number of meanvals chosen (currently 2000).

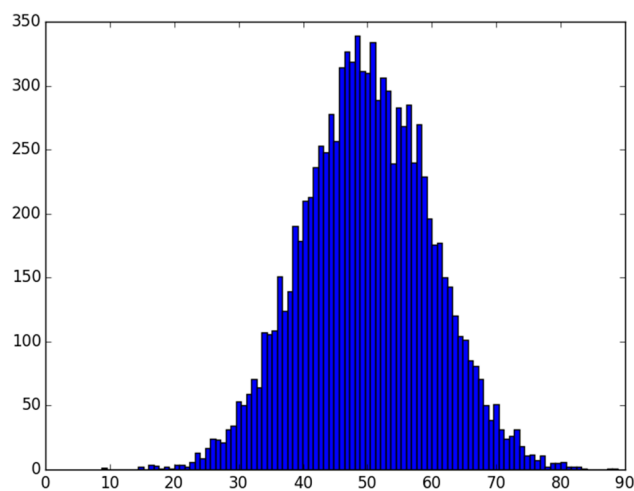
Central limit theorem

Here is an example that simulates a normal distribution and histograms the data. Notice how the mean value and the spread correspond to the values specified.


```

1. """
2. filename: 07 central limit
3. Created on Tue Apr 12 22:55:18 2016
4. @author: Ryan Smith
5. simulate central limit theorem
6. """
7. # import required packages
8. import numpy as np
9. import matplotlib.pyplot as plt
10. import random
11.
12. # generate a population of 10,000 normally distributed random numbers
13. # with mean = 50 (that's mu) and standard deviation = 10
14. t_pop = []
15. mu = 50
16. sigma = 10
17. pop_size = 10000
18.
19. for i in range(0,pop_size):
20.     t_pop.append(random.gauss(mu, sigma))
21.
22. std_pop = np.std(t_pop)
23. mean_pop = np.mean(t_pop)
24.
25. # print the standard deviation and the mean:
26. print "mean of the distribution = ",mean_pop
27. print "standard deviation = ",std_pop
28.
29. # plot a histogram of the population
30. plt.figure(2)
31. plt.clf()
32. plt.hist(t_pop,bins=100)
33. # save the figure as a png file. It will be saved
34. # to the same directory as your program.
35. plt.savefig("central limit.png")
36. plt.show()

```



mean of the distribution = 50.041169304

standard deviation = 9.88729719688

Notice that the mean and standard deviation turn out to be very close to what we asked for.

Defining functions

Super simple. The function needs to be defined before you try to use it.

```
1. def SquareSomeNumber(x):  
2.     return x**2  
3.  
4. a = SquareSomeNumber(4)  
5.  
6. print a
```

```
In [2]: SquareSomeNumber(3)  
9
```

This [stackoverflow post](#) summarizes how to call functions from another file. Both calling file and called file should be in the same directory, or a path should be specified..

Fitting data with a model

We have working here to bring in data to python so that we can do stuff with the data. One of those things is to try and fit a model to the data. We got a preview of this in example 3,

```
03 import_analysis_and_plot_data.py
```

If you have some `x_values` column and `y_values` column, then

```
np.polyfit(x_values, y_values,1)
```

will give you the slope and intercept. In other words,

`p[0]*x_values+p[1]` , or `p[0]` is your slope and `p[1]` is your y-intercept.⁴

The fit should be reasonably close to your data, if your data is a line – in this case, the last input to the `polyfit` function (the “1”) told it to fit a first order polynomial to the data. A “2” would mean fit a quadratic, etc.

Try this code:

```
1. """
2. filename: 08 fit a line to data.py
3. Created on Wed Apr 27 15:33:54 2016
4.
5. @author: Ryan Smith
6. """
7. # import libraries
8. import numpy as np
9. import matplotlib.pyplot as plt
10.
11. #####
12. # user-modified area:
13. filename= 'testData.csv'
14. xaxis_label = 'time (ms)'
15. yaxis_label = 'signal (uV)'
16. title = 'plot of my interesting data'
17. #####
18.
19. #####
20. ## defining my data #####
21. #####
22. data = np.genfromtxt(filename,delimiter=',')
23.
24. x_values = data[:,0]
25. y_values = data[:,1]
26.
27. # this test data is a quadratic - sqrt to make it a line...
28. y_values = np.sqrt(y_values)
29. # purposefully make the values be not exactly a straight line:
30. y_values[3] = y_values[3]*1.1
31. y_values[6] = y_values[6]*0.9
32.
33.
34. #####
35. ## analysis of data #####
36. #####
```

```

37. # fit a polynomial to the data
38.
39. # reminder of the inputs for polyfit:
40. #numpy.polyfit(x, y, deg, rcond=None, full=False, w=None, cov=False)
41.
42. p,cov = np.polyfit(x_values, y_values, 1,cov=True)
43. polyfxn = p[0]*x_values+p[1]
44. print "m*x + b : m = ", p[0],", b = ",p[1]
45.
46. # calculate error on fit parameters: the covariance
47. # diagonal elements are what you want
48. # http://stackoverflow.com/questions/27757732/find-uncertainty-from-polyfit
49. print sqrt(diag(cov))
50. print "m error = ",sqrt(diag(cov))[0]
51. print "b error = ",sqrt(diag(cov))[1]
52.
53. #####
54. ## plotting #####
55. #####
56. plt.figure(1)
57. plt.clf()
58. plt.plot(x_values,y_values,'.-b',label='original fxn')
59. plt.plot(x_values,polyfxn,'.-r',label='poly fit fxn')
60. plt.xlabel(xaxis_label,fontsize=15)
61. plt.ylabel(yaxis_label,fontsize=15)
62. plt.grid()
63. plt.title(title,fontsize=20)
64. plt.legend(frameon=False,loc=2)
65.
66. plt.savefig('polyfit data.png')
67.
68. # Info about legend:
69. # http://matplotlib.org/api/legend\_api.html

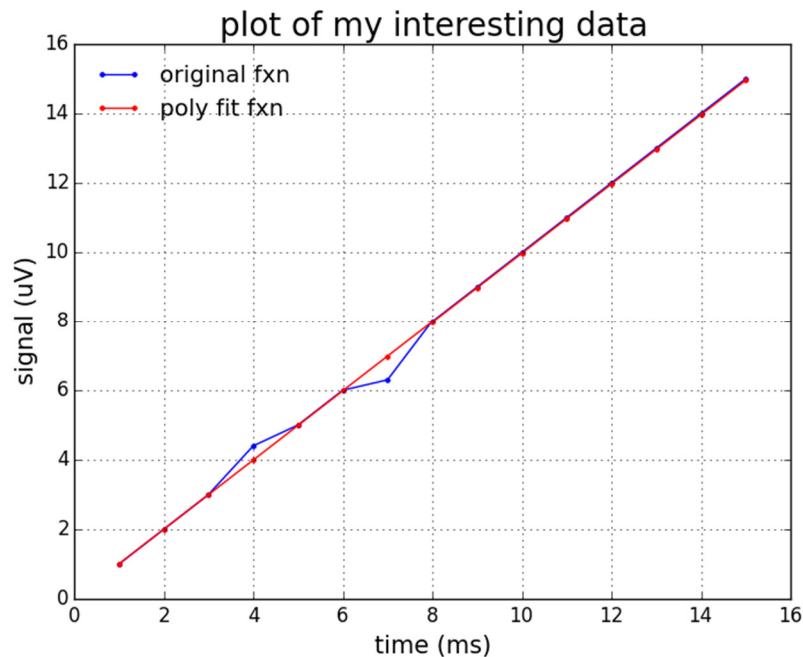
```

outputs:

```

m*x + b : m = 0.996785714286 , b = 0.00571428571429
m error = 0.0144274623465
b error = 0.131176235272

```



It's awesome that we can fit any data just with one line of code. That line of code above is **line 42**:

```
p,cov = np.polyfit(x_values, y_values, 1,cov=True)
```

The `p` is the array that contains the slope and intercept (`m` and `b`). The `cov` is the covariance matrix. Without getting too deep into the [statistics theory](#), the diagonal components of the covariance matrix are the variance (σ^2) for `m` and `b`, respectively. This simple technique then gives us some error bars in our fit parameters - especially helpful if these fit parameters are the physical parameters that we are after that determine the way that a measured curve looks!

Note: It is often helpful to cast your data into a linear format if you are expecting a relationship between two variables in your model. For example, the period of a pendulum is related to its length by:

$$T = 2\pi \sqrt{\frac{L}{g}} \quad \text{or,} \quad L = \frac{g}{4\pi^2} T^2$$

So instead of plotting T as y-axis and L as the x-axis and trying to fit a square root function to your data, you might be better off plotting L as the y-axis, and T^2 as the x-axis (as in the second version of the relation above), then fitting a line - the slope will be $\frac{g}{4\pi^2}$. That will allow you to extract the acceleration from gravity, or “little g .”

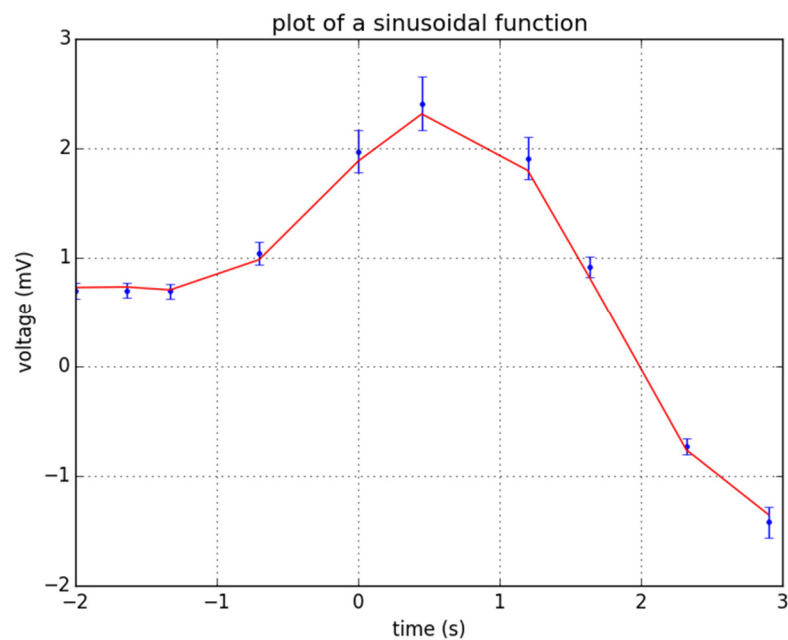
Sometimes life isn't so simple, and you have a combination of polynomials and other functions like sinusoidal or other even more exotic functions. No worries! `curve_fit` from the `scipy` library has you covered:

```
1. """
2. filename: 09 fitting_with_model.py
3. Created on Wed Apr 13 09:21:43 2016
4.
5. @author: ryansmith
6. """
7. #import the libraries we need:
8. import numpy as np
9. from scipy.optimize import curve_fit
10. import matplotlib.pyplot as plt
11.
12. # define our data
13. xdata = np.array([-2,-1.64,-1.33,-0.7,0,0.45,1.2,1.64,2.32,2.9])
14. ydata = np.array([0.699369,0.700462,0.695354,1.03905,1.97389,2.41143,1.91091,0.919576,-
    0.730975,-1.42001])
15.
16. ydata_err = ydata/10
17.
18. # this is our fit function
19. def func(x,p1,p2):
20.     return p1*np.cos(p2*x) + p2*np.sin(p1*x)
21.
22. # use the fit function to find the best parameters that match our data
23. popt, pcov = curve_fit(func, xdata, ydata,p0=(1.0,0.2))
24.
25.
26. # evaluate the fit fxn by putting in the extracted parameters from the curve fit.
27. p1 = popt[0]
28. p2 = popt[1]
29. print "fit function -> p1*np.cos(p2*x) + p2*np.sin(p1*x)"
30. print "p1 = ",p1
31. print "p2 = ",p2
32. fit_fxn = func(xdata, p1,p2)
33.
34. #####
35. ## error analysis
36. #####
37. #do some analysis: figure out the residuals:
38. residuals = ydata - func(xdata,p1,p2)
39. sum_res = np.sum(residuals**2)
40. chi2 = (np.size(xdata)-2)**-1*np.sum( (residuals/ydata_err)**2 )
41. #print "sum of residuals = ",sum_res
42. print "chi^2 = ",chi2
43.
44. #####
45. ## plotting
46. #####
47. plt.figure(1)
48. plt.clf()
49. plt.errorbar(xdata, ydata, yerr=ydata_err, fmt='.')
50. #plot(xdata, ydata, '.')
51. plt.plot(xdata, fit_fxn, 'r')
52. plt.xlabel('time (s)')
53. plt.ylabel('voltage (mV)')
54. plt.title('plot of a sinusoidal function')
55. plt.grid(True)
```

```

56. plt.savefig("test.png")
57. plt.show()
58. plt.fig = plt.gcf()
59. plt.fig.canvas.manager.window.raise_()
60.
61. #####
62.
63. # optional: save as a png file
64. plt.savefig("example data with error bars and fit.png")

```



Amazingly, with a few lines of code you can determine the parameters that complex data by drawing on a powerful nonlinear fitting routine (`curve_fit`) from the `scipy` library.

χ^2 is χ^2 which is a measure of how well a model fits to the data. When $\chi^2 = 1$ then it is a good fit.

fit function -> $p1 \cdot \cos(p2 \cdot x) + p2 \cdot \sin(p1 \cdot x)$

`p1 = 1.881850994`

`p2 = 0.700229857403`

`chi^2 = 0.404594214594`

Plotting data with error bars

```
1. plt.errorbar(xdata, ydata, yerr=ydata_err, fmt='.')
```

This is a new line that we didn't encounter yet. Should be fairly self explanatory. `fmt` is the format, and can be a dot, an 'o', or many other symbols.

Random notes about using python:

- If you are typing stuff into the console and getting out nothing, like this:

```
In [2]: a=1
```

```
....:
```

you might be in the Ipython console. check your tabs to make sure you are running in the python console, and restart the kernel if needed. If all else fails, save your scripts and restart Spyder.

- Python starts counting with 0. A little strange, but this is related to the ways that binary numbers work. So the 0th element is what you might think of as a first one in a list or array.

for example:

```
In [2]: np.arange(3,5,0.1)[0]
```

```
3.0
```

```
In [3]: np.arange(3,5,0.1)[1]
```

```
3.1
```

- Python cares about caps. for example:

```
import numpy as np
```

works just fine but

```
Import numpy as np
```

will throw errors!

- Python complains about indentations. This matters.

-

A note about division!

A big bummer about python 2 (as compared with python 3) is that its division is weird.

Try `1/2`. You get zero. This is no good for us scientists!

Workaround is to import division by either entering the following in the cmd prompt or adding this line to the top of your script:

```
from __future__ import division
```

now


```
In [2]: 1/2
0.5
In [2]: 1/3
0.3333333333333333
```

All of the script code was formatted for MS Word using this tool:

<http://www.planetb.ca/projects/syntaxHighlighter/>

Updates in this file compared with previous versions

v2.0 (Updated Aug 2, 2016):

- Fixed every line (I hope) so that we are now consistent with the proper procedure for importing libraries:

```
1. # import relevant packages
2. import numpy as np
3. import matplotlib.pyplot as plt
```

- Checked all reference files and cleaned up for consistency, checked to ensure that the *.py file corresponds to the lines of code listed here (re-formatted every file into this document). There are now 9 documents, 09 fitting_with_model.py being the last in this series.
- Next steps: more on chi² with clear explanation (can use this last example), determine what students need/want for their data analysis, propagation of errors (?), np.interpolate (and the one that Isaac was using that doesn't need monotonically increasing data)
- This module seems relatively complete as an introduction to python for scientific purposes.