Python Primer

1 What is python?

Python is an interpreted, object oriented, high-level programming language. An interpreted programming language is different from a compiled language in a number of ways, but in practice it means that you need to define all of the classes and functions at the top of the code, before you use them. In compiled languages some things can appear out of order, because the compiler reads through the whole source code before making the compiled program. Interpreted languages are read line by line by the interpreter, so things cannot be out of order in the code.

1.1 Shell vs Scripts

There are two ways to use python, in shell mode or in script mode. In shell mode, you simply type the command **python** into a terminal, which will print some details about the version of python you have installed, the date, and the commands for how to get help, credits, and licensing information. The terminal will show ">>> _", which means you can now type in python commands and the interpreter will execute those commands. For example:

```
username@computername:~/home$ python
Python 2.7.12 (default, Jul  1 2016, 15:12:24)
[GCC 5.4.0 20160609] on linux2
Type ``help'',``copyright'', ``credit'' or ``license'' for more information.
>>> 2+2
4
>>> print "this is how you Print strings to the terminal"
this is how you Print strings to the terminal
>>> a=2
>>> b=3
>>> a+b
5
>>> a*b
6
>>> a**b
8
>>> exit()
```

The second way to use python is in script mode. You write a text file containing all of the python commands you want python to execute, in the order that you want them done. After you save this file, which is called a script, you type **python** into a terminal with your script filename as the first argument, and the python interpreter reads the script and executes it line by line. For instance, the script might be:

```
a=2
b=3
print a+b, a*b, a**b # prints a plus b, a times b,
                     # and a to the exponent b
```

Saving this as "python_script", and then running it in a terminal gives:

```
username@computername:~/home$ python python_script
5 6 8
```

Note, the # denotes a comment in python.

1.2 Modules and functions

The python interpreter knows a few commands on its own, such as arithmetic operations, the **print** command for printing strings to the terminal, if-then statements, for and while loops, and various other basic programming operations. However, often you need more sophisticated tools. There are two ways to get python to use these tools: import them as modules, or define them yourself in the script.

Modules are like prewritten scripts, containing functions, classes and methods. In order to get the python interpreter to use these, you need to import the module at the beginning of your script. For instance, if you want to use the exponential function, you need to use the **math** module:

```
import math

print "e=", math.exp(1)
```

The out put is:

```
username@computername:~/home$ python python_script
e= 2.71828182846
```

Notice, if you want to use a function from a module, you need to type the name of the module, a peroid, and then the name of the function. You can also give the module a different name:

```
import math as m
print "e=", m.exp(1)
```

This produces the same output. If you only want one specific function from a module, you can import just that function. In this case, you do not need to include the name of the module when you use the function:

```
from math import exp
print "e=", exp(1)
```

You can also define your own functions:

```
def taylor_exp(x):
  y = 1 + x + (x**2)/2 +(x**3)/(2*3) + (x**4)/(2*3*4) + (x**5)/(2*3*4*5)
  return y

print "e is approximately", taylor_exp(1.0)
```

This gives:

```
username@computername:~/home$ python python_script
e is approximately 2.71666666667
```

Notice, the argument of the function **taylor_exp** is 1.0, not just 1. If you give the function an integer, it will return an integer, so if you give the function 1, it will simply return 2. We can fix this by changing the function:

```
def taylor_exp(x):
  x = float(x)
  y = 1 + x + (x**2)/2 +(x**3)/(2*3) + (x**4)/(2*3*4) + (x**5)/(2*3*4*5)
  return y

print "e is approximately", taylor_exp(1)
```

In this way, the terms involve a floating point number operation, so the result will be a float. Since python is interpretive, it does not know we want a floating point output unless we tell it to give us that somehow. Another important feature of the above example is the indentation. This is not just a stylistic choice, python uses indentation to delimit where sections of the code begin and end. After the **return y** command, the following code begins at the left margin again, whereas within the function the code is indented. Python knows that all of the indented code belongs to the function **taylor_exp**, and that the function ends once the code is no longer indented, at the print statement.

1.3 Loop examples

Instead of typing out all of the terms in the Taylor series, we can write a for loop that calculates the factorial in the denominator and then the corresponding term in the series. For example, to 9^{th} order:

```
def taylor_exp(x):

  den = 1
  y = 1.0

  for n in range(1,10):
    den = den*n
    y = y + 1.0*(x**n)/den
    print y, n
  return y

print "e is approximately", taylor_exp(1)
```

Again, the code within the for loop is further indented, python can tell that the **return y** statement does not belong to the loop.

Using a while loop, we can check what order the Taylor expansion converges to the math module value: from math import exp

```
from math import exp

def taylor_convergence_order(x):

  den = 1
  y = 1.0
  n=0

  while y - exp(1) < 0:
        n += 1
        den = den*(n)
        y = y + 1.0*(x**n)/den
  return n

print "Converges when n = ", taylor_convergence_order(1)
```

The output is 17.

2 Numpy operations

Numpy is a module that allows you to create and manipulate multi-dimensional arrays. A simple example involving a 1D array of numbers is

```
import numpy as np

a = np.array([1, 2, 3])  # Create a 1D array
print type(a)            # Prints "<type 'numpy.ndarray'>"
print a.shape            # Prints "(3,)"
print a[0], a[1], a[2]   # Prints "1 2 3"
a[0] = 5                 # Change an element of the array
print a                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])   # Create a 2D array
print b.shape                     # Prints "(2, 3)"
print b[0, 0], b[0, 1], b[1, 0]   # Prints "1 2 4"
```

Sub-arrays within an array can be accessed using a colon before an index [: a], to indicate all elements up to that index, after an index [a :], to indicate all elements after that element, and between to indexes [a : b], to indicate all elements between a and b. This is called slicing. For example:

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print a[0, 1]   # Prints "2"
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print a[0, 1]   # Prints "77"
```

Numpy array pointwise arithmetic is very intuitive:

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print x + y

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print x - y

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print x * y

# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print x / y

# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print np.sqrt(x)
```

To take inner products between arrays, or to do matrix multiplication, we use the **dot** method:

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print v.dot(w)
print np.dot(v, w)

# Matrix / vector product; both produce the rank 1 array [29 67]
print x.dot(v)
print np.dot(x, v)

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print x.dot(y)
print np.dot(x, y)
```

You can also sum the elements along rows and columns of an array:

```
import numpy as np

x = np.array([[1,2],[3,4]])

print np.sum(x)  # Compute sum of all elements; prints "10"
print np.sum(x, axis=0)  # Compute sum of each column; prints "[4 6]"
print np.sum(x, axis=1)  # Compute sum of each row; prints "[3 7]"
```

A full list of numpy functions can be found at:
`https://docs.scipy.org/doc/numpy/reference/routines.math.html`

3 Fitting and plotting data

Using the **pyfits** module, you can import data from a FITS file. Once the data has been imported, it can be converted into a numpy array to make manipulations of the data easier:

```
import pyfits
import numpy as np

filename = "data_file.dat" # Specifies the name of the file

data=pyfits.open(filename)[0].data # Imports data

data_array = np.array(data) # Converts data to numpy array
```

If the file is not in the current directory, the filename that you give to the **pyfits.open** function must include the path to the file. This only works if your data file is a FITS file, with a proper header. If you have an ASCII file, with just columns and rows of numbers, you can load it into a numpy array as follows:

```
import numpy as np

filename = "data_file.dat" # Specifies the name of the file

data = np.loadtxt(filename) # Stores data in numpy array
```

Now, suppose we want to fit the data with a function, by minimizing the χ^2. We can do this using **curve_fit** from the **scipy.optimize** module. First we need to define the fit function as a function of one independent variable, and a set of parameters that are to be optimized. In this example, we will use sinusoidal data, so the fit function is:

$$f(\theta; A, V, \theta_c, P) = \frac{A}{2}\left(1 - V\sin\left(\frac{\theta - \theta_c}{P}\right)\right)$$

Here the independent variable is the angle θ, and there are four parameters: A, V, θ_c, and P. The code defining the fit function is:

```
import numpy as np
import math as m

# Defines fit function, angles are in degrees,
# the function converts them to radians
def fitfunction(x, A, V, bc, P):
    return A/2*(1-V*np.sin((x-bc)/P*(m.pi/180.0)))
```

Now we need to import that data from our file. In this case, we will give the filename as an argument for the script. When we run the script, the command will be **python scriptname filename**, where filename is the name of the data file. The code that defines the fit function and loads the data is:

```
import sys
import pyfits
import numpy as np
import math as m

# Defines fit function, angles are in degrees,
# the function converts them to radians
def fitfunction(x, A, V, bc, P):
    return A/2*(1-V*np.sin((x-bc)/P*(m.pi/180.0)))

data = np.loadtxt(sys.argv[1]) # Stores data in numpy array
```

The arguments for the script are stored in **sys.argv**. The first element for the argument array, **sys.argv[0]**, is actually the script name.

Next we set initial values and limits for the fit parameters. We need to choose reasonable values for the fit to converge.

```
import sys
import pyfits
import numpy as np
import math as m

# Defines fit function, angles are in degrees,
# the function converts them to radians
def fitfunction(x, A, V, bc, P):
    return A/2*(1-V*np.sin((x-bc)/P*(m.pi/180.0)))

data = np.loadtxt(sys.argv[1]) # Stores data in numpy array

init_vals = [10.8*10**5, 1.1*10**-1, 90, 0.5] # Initial values of fit
limits = [[0.,0.0,0,0.49],[np.inf,1.0,360,0.51]]
```

The values are given in the same order that the parameters appear in the list of arguments for the fit function. Note, the first argument of the fit function must be the independent variable. The lower and uppers bounds for the fit parameters are given in a 2D array.

We will use th **curve_fit** function in the following form:

```
( curve_fit(polwave, data[:,0], data[:,1], p0=init_vals,
            sigma= dat[:,2], bounds=limits) )
```

The arguments are:

1. **fitfunciton**: tells **curve_fit** to use the function we defined above
2. **data[:,0]**: the independent variable, angles in this case, which are given in the first column of our data file
3. **data[:,1]**: the dependent variable, given in the second column of our data file
4. **p0=init_vals**: tells **curve_fit** to use our initial values for the fit parameters
5. **sigma=data[:2]**: the uncertainties in the dependent variable, the third column in the data file
6. **bounds**: gives bounds for the fit parameters, for instance they all must be positive

curve_fit will return two things: first a 1D array of best-fit values for the parameters, second a 2D array called the covariance matrix, which indicates the uncertainty in the fit parameters.

```
from scipy.optimize import curve_fit
import sys
import pyfits
import numpy as np
import math as m

# Defines fit function, angles are in degrees,
# the function converts them to radians
def fitfunction(x, A, V, bc, P):
    return A/2*(1-V*np.sin((x-bc)/P*(m.pi/180.0)))

data = np.loadtxt(sys.argv[1]) # Stores data in numpy array

# Initial values of fit, and limits for fit paramteres
init_vals = [10.8*10**5, 1.1*10**-1, 90, 0.5]
limits = [[0.,0.0,0,0.49],[np.inf,1.0,360,0.51]]

# Performs fit
best_vals, covar = ( curve_fit(fitfunction, data[:,0], data[:,1],
                    p0=init_vals, sigma= data[:,2], bounds=limits) )

chisq=0
A=best_vals[0]
V=best_vals[1]
c=best_vals[2]
P=best_vals[3]

# Calculates chi squared
for i in range(len(data)):
    chisq += ( ((float(fitfunction(data[i][0],A,V,c,P))-float(data[i][1]))/
                float((data[i][2])))**2 )

# Calculates reduced chi squared
chisq = chisq/(len(data)-4)

# Prints reduced chi squared
print "chisq =", str(chisq)
```

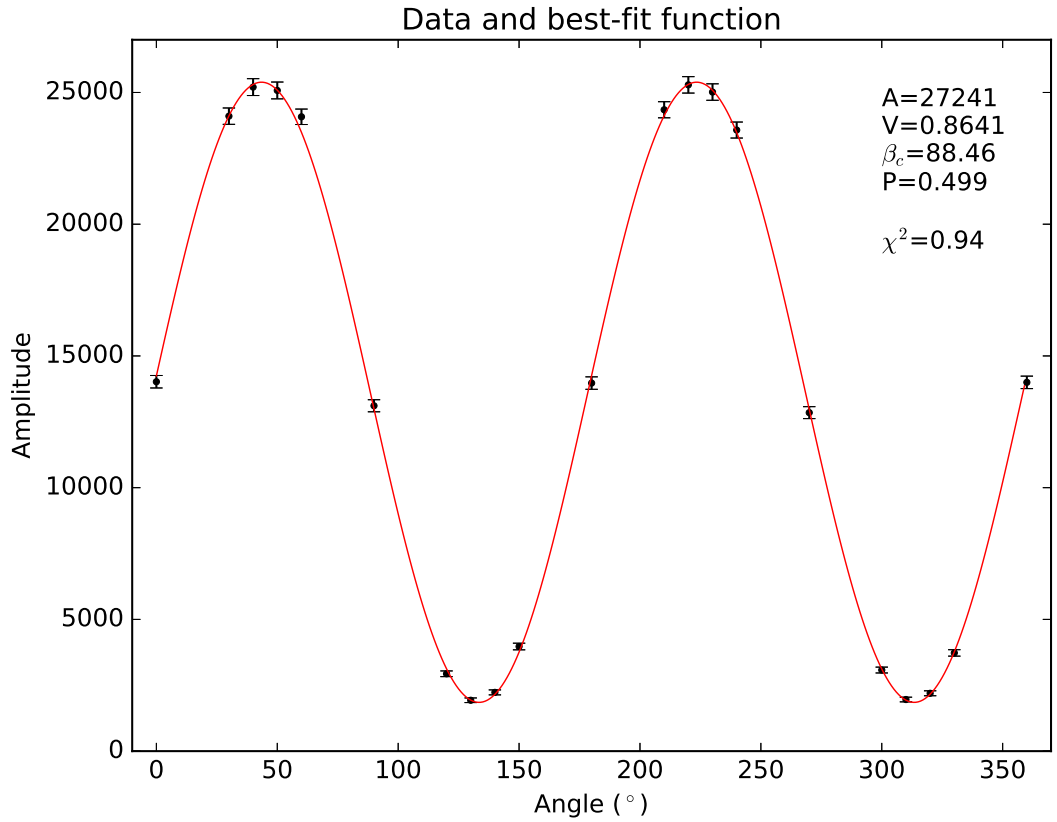
The last part calculates and prints the reduced χ^2 for the fit.

The final task is to make a plot of our data with the fit function for comparison. To do this we use the **matplotlib.pyplot** module. Below is the whole script that imports the data, performs the fit, and graphs the data with the fit function:

```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import sys
import pyfits
import numpy as np
import math as m

# Defines fit function, angles are in degrees,
# the function converts them to radians
def fitfunction(x, A, V, bc, P):
    return A/2*(1-V*np.sin((x-bc)/P*(m.pi/180.0)))

data = np.loadtxt(sys.argv[1]) # Stores data in numpy array

# Initial values of fit, and limits for fit paramteres
init_vals = [10.8*10**5, 1.1*10**-1, 90, 0.5]
limits = [[0.,0.0,0,0.49],[np.inf,1.0,360,0.51]]

# Performs fit
best_vals, covar = ( curve_fit(fitfunction, data[:,0], data[:,1],
                    p0=init_vals, sigma= data[:,2], bounds=limits) )
chisq=0

A=best_vals[0]
V=best_vals[1]
c=best_vals[2]
P=best_vals[3]

# Calculates chi squared
for i in range(len(data)):
    chisq += ( ((float(fitfunction(data[i][0],A,V,c,P))-float(data[i][1]))/
                float((data[i][2])))**2 )

# Calculates reduced chi squared
chisq = chisq/(len(data)-4)

# Defines horizontal axis plotting range
x = np.array(range(36000))/100

# Creates plot of data with error bars
plt.errorbar(data[:,0],data[:,1],data[:,2], fmt='.', color='0')

# Plots best-fit function
plt.plot(x, fitfunction(x,A,V,c,P),color='r', linewidth='0.7')

# Creates labels for plot
plt.ylabel('Amplitude')
plt.xlabel('Angle ($^\circ$)')
plt.title('Data and best-fit function')

# Customizes axes ranges (x1,x2,y1,y2)
plt.axis([-10, 370, 0, 27000])

#Complicated string format that displays the fit parameters
#and the reduced chi squared on plot
(plt.text(300, 19000, 'A='+str(int(A))+'\nV='+str(V)[:6]+'\n'+r'$\beta_c$='+
  str(c)[:5]+' \nP='+str(P)[:5]+'\n \n'+r'$\chi^2$='+str(chisq)[:4]))

plt.show()
```

Using the following data, the script gives the plot below:

x	y	σ_y
0	14021	236.821
30	24101	310.49
40	25204	320.025
50	25077	319.23
60	24077	293.101
90	13109	228.989
120	2936	108.37
130	1931	87.8865
140	2230	94.446
150	3971	126.032
180	13970	236.389
210	24344	305.575
220	25289	311.699
230	25016	312.512
240	23574	303.803
270	12848	226.698
300	3078	110.959
310	1959	88.521
320	2196	93.723
330	3728	122.115
360	13997	236.618



To save the plot automatically, use:

```
plt.savefig('filename.eps', format='eps')
```

You can use other formats, such as pdf, svg, or png. For the matplotlib homepage visit: `http://matplotlib.org/`

4 Resources

4.1 General python

https://www.python.org/
https://en.wikibooks.org/wiki/A_Beginner%27s_Python_Tutorial
https://docs.python.org/3/tutorial/
https://www.tutorialspoint.com/python/
http://www.learnpython.org/

4.2 numpy

http://www.numpy.org/
https://docs.scipy.org/doc/numpy-dev/user/quickstart.html
http://cs231n.github.io/python-numpy-tutorial/
http://www.python-course.eu/numpy.php

4.3 matplotlib

http://matplotlib.org/
https://www.labri.fr/perso/nrougier/teaching/matplotlib/
http://www.scipy-lectures.org/intro/matplotlib/matplotlib.html
https://pythonprogramming.net/matplotlib-python-3-basics-tutorial/