Introduction to Python for Data Assimilation

1) Why Python?

Python is a next-generation computer language that is rapidly becoming one of the most popular and widely used. It is ubiquitous in many scientific communities. It is also *free*, both in the sense of cost and in the sense of its license (it is distributed under the Gnu Public License). These means you can *always* install Python on your computer and send code to other people knowing that they have no restrictions on being able to run that code. This is quite unlike tools such as Matlab where both parties need to have paid for copies of the software.

A key objective in teaching Python is to provide a tool that you will always be able to use (and not just for scientific computing). Unlike, say Matlab, Python is a fully fledged programming language that can be used for virtually any computer based task. Another very good reason for using Python is that it is extremely well documented. There are countless official and unofficial documentation sites. A great place to start is here:

http://docs.python.org/tutorial/

► In your own time you should work through the following pages and make sure you understand everything:

► http://docs.python.org/tutorial/introduction.html
► http://docs.python.org/tutorial/controlflow.html

2) Fundamental concepts:

Most programming languages have the following components:

- Variables
- Keywords
- Operators
- Delimiters
- Comments
- Functions

This list is a little over simplified, but it serves our purposes.

2.1) Variables

Variables are used to store data. You can give them any arbitrary name as long as it contains only alphanumeric characters and the underscore '_' character. In addition it should not start with a numeric character or, unless you know what you're doing, the underscore character. Variables names cannot contain the space character or punctuation. Finally they cannot have the same name as a Python keyword (see below).

Variables come in a number of different flavours, and in languages like Python it is even possible to create new variable types. For our purposes however we will mainly encounter the following:

- Integer numbers (ints)
- Floating point numbers (floats)
- Strings
- Lists

Variables only come into existence when you assign a value to them using the '=' operator and Python decides what type of variable to create by examining the assignment statement.

► Explore integers in the interactive shell as follows:

```
>>> a=6
>>> print a
>>> b=4
>>> print a,b
>>> print a+b
>>> c=a+b
>>> print c
```

► The following explores the relationship between ints and floats:

```
>>> a=9.0
>>> b=2.0
>>> print a,b,a*b,a/b
>>> a=0
>>> b=2
>>> print a,b,a*b,a/b
```

What do you notice about the difference in the use of integers and floating point numbers?

Strings are variables that store collections of characters that are used to form text. They are enclosed in single or double quotes. In practice it doesn't mater which although there are some aspects of good style surrounding this (that we won't go into in this class). For our purposes we will not need to do any complex manipulation of strings; they will mainly be used for labelling plots and so on.

► Try the following:

```
>>> a=’data’
>>> b=’assimilation’
>>> print a,b
>>> print “data”,b
```

Finally, lists will be a very important variable type to us. They quite literally contains lists of any other variables, even other lists. Each list item can be retrieved from the list by referencing it using an integer number. Note that the first item in the list has the index zero.

► Try the following:

```
>>> a=[6,5,8,3,5]
>>> print a
>>> print a[0]
>>> print a[1]
>>> print a[-1]
>>> b=7
>>> a=[6,5,b,3,5]
>>> print a
```

Although the above example has used only integers you could just as easily use floating point numbers or a combination of the two (or any other variable type for that matter). There are lots of neat things that can be done with lists to simplify programming tasks. You can learn more about them here:

► http://docs.python.org/tutorial/introduction.html#lists
► http://docs.python.org/tutorial/datastructures.html#more-on-lists

There are a couple of additional things about lists you need to know now:

► A blank list is created by assigning empty delimiters:

```
>>> a=[]
```

You can also do this to clear an existing list.

► Add items to the end of any list using ‘append()’:

```
>>> a=[]
>>> a.append( 6.0 )
>>> print a
>>> a=[6,5,8,3,5]
>>> a.append( 99.0 )
>>> print a
>>> a=[]
>>> print a
```

Technically ‘append()’ is a *method* of the class type ‘list’. For our purposes we can think of it as a function that is attached to a list and operates on that list (this is a very crude definition of what a method is).

► Finally it is possible to determine the number of items in a list using the ‘len()’ function:

```
>>> a=[]
>>> a.append( 6.0 )
>>> print a, len( a )
>>> a=[6,5,8,3,5]
>>> a.append( 99.0 )
>>> print a, len( a )
>>> a=[]
>>> print a, len( a )
```

2.2) Keywords

Keywords are reserved by Python for specific tasks. You cannot give anything else the name of one of Python’s keywords.

http://docs.python.org/release/2.3.5/ref/keywords.html

Keywords such as ‘for’ and ‘if’ enable control of the flow of a Python program. We will explore these in a short while.

2.3) Operators

Common examples of operators are ‘=’ (the assignment operator), the maths operators such as ‘*’, ‘/’, ‘+’ and ‘-’ that perform the basic mathematical operations, the inequality operators (e.g. ‘>’ the greater than operator) and equality operator (‘==’, note the double equals sign) that are used in control-flow statements.

A good introduction is here:

► http://en.wikibooks.org/wiki/Python_Programming/Operators

2.4) Delimiters

The delimiters are used to indicate the indexes of lists ‘[]’, bound the arguments of functions and methods ‘()’ and to group mathematical expressions. A full list is here:

http://docs.python.org/release/2.5.2/ref/delimiters.html

The following example illustrates grouping of mathematical expressions:

```
>>> print 6/2+1
>>> print 6/(2+1)
```

Do not continue if you don’t understand why these are different: ask!

2.5) Comments

Comments are ignored by Python. Any character *after* the hash (‘#’) character and on the same line is ignored. You can write whatever you want here. They are of little value in the interactive shell but you can use them in programs to leave notes about what each bit of your code does. This is extremely important and you should get into the habit of using comments from the outset. Another use of comments is to temporarily remove a line of code (or several lines of code) from execution.

In addition any text directly following the start of function and enclosed in triple quotes is taken as an extra long comment, called a ‘doc string’. It is also the text that the ‘help’ function accesses.

► http://en.wikibooks.org/wiki/Python_Programming/Source_Documentation_and_Comments

2.6) Functions

Think of functions as small reusable bits of Python code that do specific jobs. There are a number of built-in Python functions, such as ‘len()’ that we saw earlier, and a vast number of functions that come with different modules. Perhaps more importantly it is easy to create your own functions and store them to be reused whenever you need. This is a corner stone of good programming practice.

An overview of how functions work can be found here:

► http://en.wikibooks.org/wiki/Python_Programming/Functions

Functions are preceded by the ‘def’ keyword. This is followed by the name of the function which must follow the same conventions as naming variables (see above) followed by a list of arguments that are delimited by parentheses, ‘()’. Directly after the argument list is the ':' delimiter. Python statements after the ':' should be indented to the same level (normally 2 or 4

spaces) to indicate that they are part of the function. It is common for functions to return some value using the ‘return’ keyword.

Indentation often causes some confusion at first, but actually the rules for it are very simple. Indentation is used to tell Python which bits of code belong to a function or a control-flow statement (as we will see later). In short, after each use of the 'def' keyword or a control-flow keyword (e.g. 'if' or 'for') the code should be indented by a consistent amount (typically 2 or 4 spaces). As soon as the indentation is removed (i.e. back to the level of the 'def' keyword) the code is considered to be outside of that function.

Variables defined inside a function belong to that function and will not effect the value of variables with the same name that are outside of that function. This is best illustrated by example.

► In the interactive shell type:

```
>>> def foo( a ):
        b=a


>>> b=3
>>> foo( 7 )
>>> print b
```

What is the value of 'b' that is printed? Make sure you understand why the function 'foo' has not changed the value of 'b' and ask if you don't understand. Note that the function 'foo' is completely useless for all practical purposes, it is only used here to illustrate the scope of variables defined inside functions. (Also note that we have learned something new: it is possible to define function inside the interactive shell.)

3) Control-flow syntax

So far we have not described how to make decisions within a program. This is the job of control flow statements, which include the keywords ‘if’ and ‘for’. Being able to conditionally execute certain portions of code and iterate over the same pieces of code with changes in variable values is fundamental to almost all programs.

3.1) The ‘if’ statement

The 'if' statement conditionally executes a section of code. The conditions can be quite varied but we will almost always be dealing with equalities and inequalities. The code to be subject to the conditional execution should be indented underneath the initial 'if' statement. It is also possible to use an 'else' statement to specify a block of code that should be executed *only* if the original condition is not met.

► Here is a simple example of a function using an 'if' statement:

```
>>> def diffAbs( a, b ):
        if a > b:
            return a-b
        else:
            return b-a


>>> diffAbs( 7,8 )
1
>>> diffAbs( 8,7 )
1
>>>
```

This function ensures that the smallest number is always subtracted from the biggest number and not the other way around (thus avoiding a negative result). Note that in practice the same result can be achieved more efficiently using Python's built-in 'abs()' function. Also it would be better to put functions into their own file (using a text editor) so they can be re-used and thoroughly commented.

Notice the way in which indentation works: the 'return' statements are indent 4 spaces because they are inside the function and an additional 4 spaces because they are inside a block of code (albeit a one line long block of code in this case) that will be executed conditionally dependant on the 'if' statement. You can also nest if statements inside one another, and the level of indentation would need to be increased with each nesting. There is also an 'elif' keyword that allows additional conditions to be tested if previous ones have not been met.

► Try out the following:

```
>>> def biggerThan( a ):
        if a > 7:
            print a,'is bigger than 7'
        elif a > 3:
            print a,'is bigger than 3'
        else:
            print a,'is less than or equal to 3'
        print 'this print statement always happens!'


>>> biggerThan( 8 )
>>> biggerThan( 2 )
```

The following page provides a fairly accessible overview of 'if' statements:

► http://en.wikibooks.org/wiki/Python_Programming/Conditional_Statements

3.2) The ‘for’ loop

A 'for' loop allows a section of code to be executed an arbitrary number of times whilst changing the value of variables inside that block of code each time. We will always encounter the following syntax:

```
>>> for var in list:
      #add some Python code (indented!)
```

The variable 'var' (which can have any valid variable name) is set to each value stored in the list variable 'list' in turn and then the indented Python code is executed for each value of 'var'. Note the use of the ':' delimiter, just like after a function definition.

► The following example requires that you have the 'biggerThan()' function already defined:

```
>>> for n in [5, 6, 2, 8, 9, 100, -99, 3.1415]:
        biggerThan( n )
```

Alternatively, the list could be generated beforehand:

```
>>> someNumbers=[5, 6, 2, 8, 9, 100, -99, 3.1415]
>>> for n in someNumbers:
        biggerThan( n )
```

There are also tools to generate lists. The Python 'range()' function is specifically designed to generate lists of integers for use in 'for' loops. However any function that returns a list can also be used.

► The following example uses the Python 'range()' function.

```
>>> def isPrime( a ):
            for i in range( 2,a ):
                if (a%i)==0:
                    print a,'is not a prime number'
                    return
            print a,'is a prime number'


>>> isPrime( 6 )
>>> isPrime( 7 )
```

► Use Pythons built-in 'help()' function to find help on 'range()'.
► Make sure you can understand the following questions:

- What is the list of numbers that 'range()' generates?
- What does the '%' operator do? (Look this up!)
- Why is there a 'return' statement inside the conditionally executed code block?

Finally, a couple of things to note: 1) A function this size or bigger should really be in a separate file (with comments!); 2) it is not a very efficient way of identifying prime numbers; 3) there is a more efficient version of 'range()' called 'xrange()', but the help text isn't quite as nice (and it generates something that is a bit more complex than a simple list).

► Read the following pages in your own time:
► http://en.wikibooks.org/wiki/Python_Programming/Loops#For_Loops

4) Importing a module

An important aspect of Python is that it has large number of modules that are provided with it, and a huge number of modules that can be added trivially. These modules extend the functionality of Python to perform specific tasks. We will use a number of common modules in almost everything we do.

► Type:

```
>>> import numpy as np
```

There are various different ways of achieving the same thing in Python but we will stick with the syntax above. We have now loaded the numpy (or ‘Numerical Python’) module and associated it with an ‘object’ that we have arbitrarily called ‘np’. We can now access any of numpy’s functionality from the ‘np’ object.

► For example numpy has a number of useful mathematical constants:

```
>>> print ‘The value of pi is’, np.pi
```

► It also has useful functions, such as trigonometry. What answer does the following give?

```
>>> np.cos( np.pi )
```

5) Getting help

For most of the widely used Python functions you can use the help command to find out more details.

► Get help on the ‘linspace’ command:

```
>>> help( np.linspace )
```

Help will not work for every command, but should be sufficient for most of what we will come across. In addition, as always, the modules are very well documented. The main modules we will use are numpy, scipy and matplotlib. The web pages for their documentation are (n.b. the scipy pages cover numpy as well):

http://docs.scipy.org/
http://matplotlib.sourceforge.net/contents.html

► There are also some very relevant tutorial pages. The following provides a good introduction to matplotlib and numpy:

► http://matplotlib.sourceforge.net/users/pyplot_tutorial.html
► http://www.scipy.org/Tentative_NumPy_Tutorial

6) Writing a simple function

So far so good, but if we want to write something more complex using the interactive shell is not very practical: we need to write a program.

► In the editor open a new window.

This will open an editor, which does not interpret the Python commands as you type them. Start by saving the blank file (call it 'mylib.py') to your work space into a new folder. Remember to save your work regularly.

► Now, inside the editor type the following code:

```
def myAbs( v ):
    ‘’’Returns the absolute value of v‘’’
    if v<0:
        return -1.*v
    return v
```

Save the file. We have written a very simple function that takes a single argument (‘v’) and prints it to the screen alongside some text.

Note: The way that Python defines the *scope* of the function is to use indentation. Any instructions that are inside the function must have the same, or deeper, level of indentation. The scope of any conditionals or loops (e.g. the 'if' statement in the above example) is handled in the same way.

► Back in the interactive shell, type:

```
>>> from mylib import *
```

► Now type:

```
>>> print myAbs( -5.3 )
```

Additional functions can be added to the mylib.py file. By building up a library of useful functions in this way you can quickly re-use code.

7) Some good practice

a) Under every function you create write a description of it in triple quotes (as above). Python will ignore this but the help function reads it in to provide it's text.

► Try:

```
>>> help( myAbs )
```

You should be thorough in your description of the function.

b) Throughout your code you should add comments. In Python anything after the '#' character is ignored. This is useful to leave messages to yourself (and someone else who comes to use your code). We haven't done this so far but we will from now on.

c) Use sensible variables names. We have been using single character names so far. These are best avoided unless they make particular sense. Instead use whole or abbreviated words. This will make your code more readable.

8) Displaying data with matplotlib

Matplotlib is a python module that has a similar syntax to Matlab's plotting functionality. Typically we only need to import the "pyplot" component of the module. This can be done as follows:

```
>>> import matplotlib.pyplot as plt
```

Here we have assigned pyplot to an object named "plt", this could have any name we wanted however, so don't be confused if examples elsewhere use a different form. Before we start plotting we need to generate some data:

```
>>> d=np.sin(np.linspace(0,np.pi*4,200))
```

This is a simple way to generate a vector that contains a sine wave. We can then plot this easily using the following commands:

```
>>> #clear any existing plots:
>>> pl.clf()
>>> #turn on interactive mode:
>>> plt.ion()
>>> #plot the data:
>>> plt.plot(d)
```

You should now have a simple looking plot of a sine wave. So far so good but there lots that can be done to improve our plot. Try the following:

```
>>> #clear any existing plots:
>>> plt.clf();
>>> #a nicer looking sine wave:
>>> plt.plot(d,'--',linewidth=3.0,color='g',label="sine wave")
>>> #a different sine wave
>>> plt.plot(d*0.5,'.', label="sine * 0.5", color='r')
>>> #add a legend:
>>> plt.legend()
>>> #label the x-axis:
>>> plt.xlabel('time')
>>> #label the y-axis:
>>> plt.ylabel('amplitude')
```

You should now have a plot that looks like the following figure:

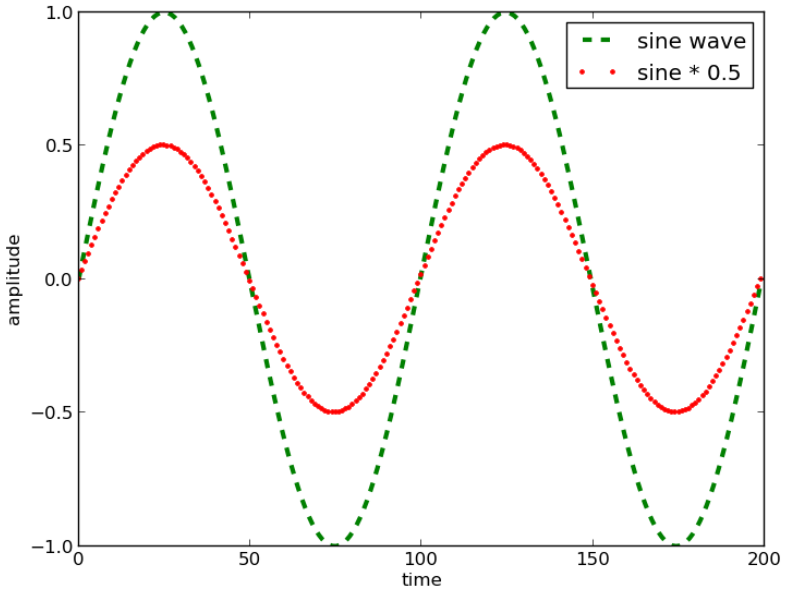

Figure 1: Some sine waves with different plotting styles

Python can also display images using the matplotlib.pyplot module. Images are represented as 2D lists or numpy matrices. Any easy way to generate some 2D data is to take the outer product of our sine wave vector:

```
>>> dd=np.outer(d,d)
```

The resulting variable, dd, is a numpy matrix. We can display it as an image by doing the following:

```
>>> plt.clf()
>>> plt.imshow( dd )
```

You should now have a figure that looks like this:

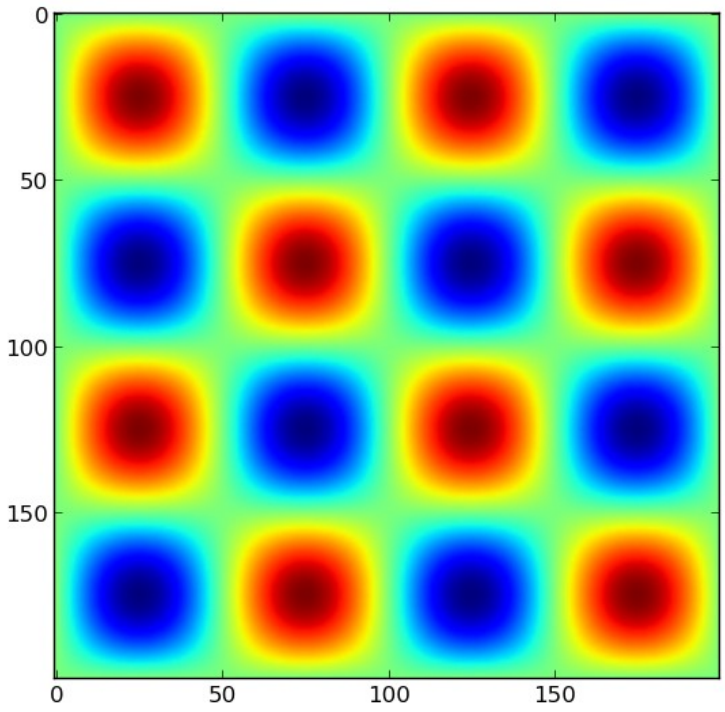

Figure 2: A matrix plotted as a 2D image

9) A more advanced example:

Here is a more advanced example, exploring the Lorenz system, that brings together many of the things we have discussed above, and adds a few new features. First create the following functions and save them:

```
def lorenz63_euler( x, y, z, p, o, b, dt=0.01 ):
  """
  Modified Euler integration of the Lorenz 63 system
  based on Matlab code by Amos Lawless (U Reading).

  x=current position in x
  y=current position in y
  z=current position in z

  p=rho parameter
  o=sigma parameter
  b=beta parameter
  dt=time step size

  returns x,y,z (as a tuple) at t+dt
  """

  dx=o*(y-x)
  dy=x*(p-z)-y
  dz=x*y-b*z

  dx1 = dt*(o*(y-x))
  dy1 = dt*(x*(p-z)-y)
  dz1 = dt*(x*y-b*z)

  dx2 = dt*(o*(y+dy1-x-dx1));
  dy2 = dt*(p*(x+dx1)-y-dy1-(x+dx1)*(z+dz1))
  dz2 = dt*((x+dx1)*(y+dy1)-b*(z+dz1))

  x=x+0.5*(dx1+dx2)
  y=y+0.5*(dy1+dy2)
  z=z+0.5*(dz1+dz2)

  return (x,y,z)
```

And the following function too:

```
def lorenz63( xInit, yInit, zInit, p, o, b, s=2000, dt=0.01 ):
  """
  Loop over the Lorenz 63 system

  x=initial position in x
  y=initial position in y
  z=initial position in z

  p=rho parameter
  o=sigma parameter
  b=beta parameter
  dt=time step size
  s=number of time steps

  returns lists x,y,z as tuple
  """

  x=[xInit]
  y=[yInit]
  z=[zInit]

  for i in xrange(s):
    #extend list size by one
    x.append(0)
    y.append(0)
    z.append(0)

    (x[i+1],y[i+1],z[i+1])=lorenz63_euler(x[i],y[i],z[i],p,o,b,dt=dt)

  return (x,y,z)
```

Now you can generate plots of the Lorenz system using slightly more advanced matplotlib than previously:

```
>>> import matplotlib.pyplot as plt
>>> from mpl_toolkits.mplot3d import Axes3D
>>> xInit=-10.
>>> yInit=-10.
>>> zInit=20.
>>> p=28.
>>> o=10.
>>> b=8./3.
>>> (x,y,z)=lorenz63( xInit, yInit, zInit, p, o, b )
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111, projection='3d')
>>> ax.plot(x,y,z)
>>> ax.set_xlabel('x')
>>> ax.set_ylabel('y')
>>> ax.set_zlabel('z')
>>> plt.show()
```

You should get a plot that looks like the one below. Once you have finished creating the Lorenz plots try the following:

- Change the values of the initial conditions (`xInit, yInit, zInit`)
- Change the values of the parameters (`o, p, b`)
- Plot x, y and z in 2D as a function of time

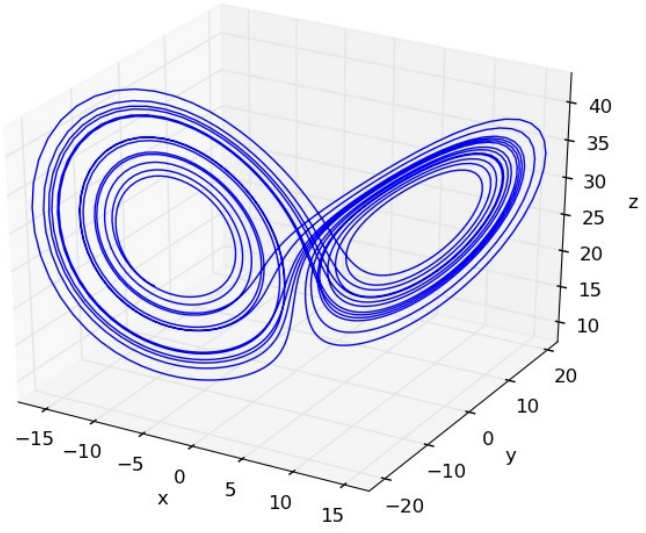

Figure 3: A 3D plot of the Lorenz 63 system

10) Further reading:

There are lots of web based Python resources, some of which have already been mentioned. Here is a brief list to provide you with some good starting points for further reading:

- http://www.ibiblio.org/g2swap/byteofpython/read/
- http://en.wikibooks.org/wiki/Python_Programming/
- http://www.brpreiss.com/books/opus7/html/book.html