PYTHON Basics

http://hetland.org/writing/instant-hacking.html

Python is an easy to learn, modern, interpreted, object-oriented programming language. It was designed to be as simple and accessible as possible with an emphasis on code readability. For chemist, it is one of the most widely used scripting languages.[1]

Python interpreter

You can start the Python interpreter from any terminal window. When you first start the interpreter program, you will see something like the following:

```
$ python
Python 2.5.1 (r251:54863, Jan 13 2009, 10:26:13)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is the Python *prompt* indicating it is waiting for a command. In programming languages, a complete command is called a *statement*.

```
>>> print "Hello"
Hello
>>> 2+3
5
>>> print "2 + 3 =",2+3
2 + 3 = 5
```

Note: the red text denotes python screen output. You can *assign* a value to a variable by putting the variable name on the left, followed by a single =, followed by what is to be stored.

```
>>> x = 1.0+2**3
>>> x
9.0
```

To execute a sequence of statements, you can create a command called a *function*. For continuation lines, a *secondary prompt* ("…") is displayed when you press the enter key. A blank line (or a secondary prompt on a line by itself) is used to end a multi-line command. As you enter the following statements, be sure to indent as shown. (You can use the tab key)

```
>>> def hello():
...     print "Hello"
...     print "Chemistry is fun!"
...
>>> hello()
Hello
Chemistry is Fun
>>>
```

Typing an end-of-file character (Control-D) at the primary prompt causes the interpreter to exit the session.

[1] http://baoilleach.blogspot.com/2008/03/python-scripting-language-of-chemistry.html

Python modules

You can also execute statements in a text file, *i.e.*, program, from a terminal window. Python source files have a `.py` suffix. For example, use the `vi` editor to create a file named "`circle.py`" containing the following statements.

```
# This function asks the user for the radius
# and prints the area of a circle

def area():
    r = input('Enter the radius of the circle: ')
    pi = 3.14
    area = pi*r*r
    print 'area = ', area

area()  # this statement calls the function
```

Note: The hash sign (#) indicates the beginning of a single-line comment statement used for program documentation.

There are two different ways to run your program. First, a python script can be executed at command line by invoking the interpreter on your application: `python [filename]`

```
$ python circle.py
Enter the radius of the circle: 4
area =  50.24
$
```

This simply executes the file and returns a UNIX command prompt. To run the program in the Python interpreter window, you must import the file:

```
$ python
>>> import circle
Enter the radius of the circle: 3
area =  28.26
```

Once the file is imported, you can run the function again by typing:

```
>>> circle.area()
Enter the radius of the circle: 6
area =  113.04
```

Python has a wide library of predefined functions that can be imported and used. For example, we can import the math library and *call* its functions using the dot-notation:

```
>>> import math
>>> print math.pi
3.14159265359
>>> x = math.atan(1.0)*4
>>> print x
3.14159265359
>>> math.cos(x)
-1.0
```

Passing & Returning Variables

Functions are an important tool in programming. As demonstrated above, the most common use of functions is to pass a function information (*e*,*g*, 1.0) or a declared variable (x) and the function returns information based on the input parameter. Note: In order to use the output of a function, you must assign a variable the return value of the function. To further illustrate this point, we can modify our circle program so that the area function takes in the value of the radius and returns the area of the circle:

```
# This function takes in the radius and returns the area of a circle

def area(r):
    import math
    area = math.atan(1.0)*4.0*r*r
    return(area)

# Main program statements

r = input('Enter the radius of the circle: ')
output = area(r)
print 'area = ', output
```

Looping Structures

If you would like to execute a python statement multiple times, you should use a loop control structure, either a *for-loop* or a *while-loop*. A for-loop is excellent for counted loops, when you know exactly how many times you want to execute your code. To iterate over a sequence of numbers the built-in function `range()` is used to automatically generate them. For example:

```
>>> def forloop():
...     for i in range(3):
...             print "the number is",i
...     print "goodbye"
...

>>> forloop()
the number is 0
the number is 1
the number is 2
goodbye
```

Unlike most programming languages, indentation is an intrinsic part of Python's syntax. The indented statements that follow the `for` line are called a *nested block*. Python understands the nested block to be the section of code to be repeated by the `for` loop. Thus, everything that is indented after the for-statement is executed three times (or however many values are in the list). All lines following the for-loop that are NOT indented are executed only once.

A while-loop is more powerful than a for-loop, because it is more flexible. Any for-loop can be written as a while-loop instead (although not every while-loop could be written as a for-loop. For example, the for-loop executed above can be re-written as:

```
>>> def whileloop():
...     i = 0
...     while i < 3:
...             print "the number is", i
...             i = i+1
...     print "goodbye"
...
>>> whileloop()
the number is 0
the number is 1
the number is 2
goodbye
```

Conditional Statements

The `if` statement executes a nested code block if a condition is true. `else` and `elif` statements allow you to test additional conditions and execute alternative statements accordingly. They are an extension to the `if` statement and may only be used in conjunction with it. Here is an example:

```
>>> def number():
...     x = input("enter a number: ")
...     if x < 10:
...             print x, "is less than 10"
...     elif x > 10:
...             print x, "is greater than 10"
...     else:
...             print "your number is 10"
...     print "goodbye"
...
>>> number()
enter a number: 4
4 is less than 10
goodbye
```

Data Types

Python has a number of available data types and associated operators and functions to manipulate them. The most intuitive are the numeric data types, such as integers and floating point numbers. A Python "list" is a series of values that are assigned by placing them within square braces and separating them by commas. Individual elements of the list are associated with an integer index starting at zero.

```
>>> numlist = [3,13,23,440]
>>> numlist[1]
13
>>> numlist[0:2]
[3, 13]
>>> len(numlist)
4
>>> numlist.append(9)
>>> numlist
[3, 13, 23, 440, 9]
```

Another important data type is a "string" which holds any combination of letters and numbers enclosed in quotes. Note: Any number defined as a string, is not treated like a numerical value but is preserved as if it were a word. Similar to lists, we can treat the character string as a single entity, or access individual components:

```
>>> dna = "CCGTAC"
>>> dna[2]
'G'
>>> dna[3:6]
'TAC'
>>> len(dna)
6
>>> strlist = dna.split('G')
>>> strlist
['CC', 'TAC']
```

File I/O

So far we have read input from the keyboard and written output to the screen. It is often more convenient to read in data from a file to analyze or use in a model calculation. Equally necessary, is the ability to save the results of a calculation to a file for later reference. Files, from a programming perspective, are not very different from files that you use in a word processor or other application. You open them to start working, read or write in them, then close them when you have finished your work. One of the biggest differences is that a program will access the file sequentially, *i.e.*, it reads one line at a time starting at the beginning. For example:

```
>>> infile = open("dna.pdb","r")
>>> line = infile.readline()
>>> line
'ATOM      1  H5T A5      1      -0.808  -8.873  -2.080\n'
>>> list = line.split()
>>> list[5]
'-0.808'
>>> xdata = []
>>> xdata.append(list[5])
>>> line = infile.readline()
>>> line
"ATOM      2  O3' A5      1       4.189  -7.682  -0.250\n"
>>> list = line.split()
>>> xdata.append(list[5])
>>> xdata
['-0.808', '4.189']
>>> line = infile.readline()
>>> list = line.split()
>>> while list[0] == 'ATOM':
...             xdata.append(list[5])
...             line = infile.readline()
...             list = line.split()
...
>>> ntot = len(xdata)
>>> ntot
229
>>> outfile = open("dna.xdata","w")
>>> for i in range(ntot):
...             outfile.write(xdata[i]+"\n")
...
>>> outfile.close()
>>> infile.close()
```

You can view your output file in new terminal window. This file simply contains the x-position of each atom in the first strand of dna identified in the file `dna.pdb`.