Week 4- Functions and Modules

This week, we introduce two new objects: functions and modules. Modules are scripts written by other whose functionality you can access in your own code. Functions are a way of creating your own blocks of code that can also be accessed by name.

Functions

Functions are objects that you make yourself. They are compact pieces of code that can be accessed by name elsewhere, preventing redundancy. They are an important part of programming in Python. Using functions forces you to think about your script as a series of modular parts, which leads to code that is more flexible, readable, and easier to use in the future. Not everyone uses them regularly. The code in Haddock and Dunn's "Practical Computing for Biologists" makes little use of functions. O'Reilly's "Bioinformatics Programming using Python," on the other hand, uses functions exclusively for the main body of their code. You will find your own way, which will likely be somewhere in between these two.

General form:

```
def name(parameter-list):
   body
```

"Parameter-list" will be zero or more comma-delimited parameters that you wish to pass to the function. The function will not work unless all the defined parameters are filled in when the function is called.

Return: Some functions do not return values, such as a function which just prints whatever values it receives. But if you want your function to return a value, you need a 'return' statement.

```
return value
```

The 'return' statement **exits the function**, and returns *value* to whatever called it.

Example

```
>>> num = 'whatever'
>>> def square(num): # function definition
…       return num**2
>>> print num # function parameter 'num' has local scope
whatever
>>> list = [1,2,3,4]
>>> for i in list:
…       print square(i) # function call
…
1
4
9
16
```

Docstrings: Comments about functions are typically done with docstrings, which, unlike comments can be seen by the interpreter.

```
def square(num):
   '''returns the square of input integer''' # docstring
   return num**2
```

Default Parameter Values: You can define a default parameter value for a function. This parameter, unlike other, can be left blank, in which case it will take on its default value. If the value is Boolean (True or False), this parameter is typically called a flag. The function below relies on the string method .count().

```
   def base_counter(seq, is_RNA=False):
   '''Prints counts for each base in a sequence. Counts 'U' if is_RNA
is True.'''
      seq = seq.upper()
      if is_RNA:
           baselist = ['A','C','U','G']
      else:
           baselist = ['A','C','U','G']
      for base in baselist:
           print base + ": %s" % seq.count(base)

   >>> seq = 'atgact'
   >>> base_counter(seq) # 'is_RNA' is evaluated as False
   A: 2
   C: 1
   T: 2
   G: 1
   >>> seq = 'augacu'
   >>> base_counter(seq, True) # 'is_RNA' is True
   A: 2
   C: 1
   U: 2
   G: 1
```

Program Flow: Your program should be a cascading set of functions. It may seem harder to make a script with functions rather than just writing it out 'globally,' but it will help you organize your code, prevent redundancy, and increase readability. It seems overwhelming at first, so write out what you want to do first, and then think about what tasks the program will have to accomplish to do this. Finally, turn these elements into functions.

Modules

Python is a popular language, and lots of people have come up with useful extensions and pieces of code, called modules. Some functionality is included in the base Python package, but others need to be called specifically into your program. Today, we'll cover some common and useful modules.

Each module has a name which, when imported, has a number of methods associated with it. The methods are accessed via the usual dot notation.

`modulename.method`

To get access to a module's functionality, you need to import it using an 'import' statement.

`import module`

You can selectively import one method from a module, or import multiple modules and methods:

```
from module import method     # import just one method from module
import module1, module2
```

Python has a number of built-in functions that are very useful, some of which are discussed in these week's Extensions. I highly recommend looking at all of them here:
http://docs.python.org/2/library/functions.html

Common Modules

These modules are not available by default and must be imported via `import` statements.

os: OS is a module that allows programmers to access the Unix operating system of the computer.

os.getcwd() : In a Python script, find out in which working directory you are located. Analogous to pwd in UNIX.

os.chdir(*path*) : From Python, change your working directory. Similar to UNIX's cd.

os.tmpfile() : Returns a temporary file object that can be read and written to, and will be erased at the end of the script.

sys: A module for access interpreter-level functionalities.

sys.argv[] : Return the list of command line argument parameters. Used to pass arguments to the script. For example, from the modularized script obs_counter2.py:

```
infile = sys.argv[1]
```

This will set the value of `infile` to the first argument provided by the user. For example:

```
./obs_counter2.py file1.txt
```

will perform the operations contained in obs_counter2.py on file1.txt

sys.stderr: Corresponds to the command line's stderr stream. This is useful because you sometimes want to capture part of the python output with the redirect (`'>>'`), but to have other parts keep on printing. Since stdout (via the 'print' statement) will be captured by the redirect, but stderr wont, you can use these two streams to separate parts of your output. Use like this:

```
sys.stderr.write("stderr string")
```

pprint: Also known as pretty print. Used to print objects in different orders and data structures.

pprint.pprint(name): Will take object called name and reformat it into a readable table. This is useful for visualizing data, as well as for piping the output of Python scripts using UNIX.

pprint.pformat(name): Will return the same, but in a plain-text representation.

itertools: I'm a big fan of this one. It offers a huge array of functions for powerful iteration and combinatorics. Here's just one example using `izip`, which is similar to the base python function `zip`, but more efficient, especially for large list objects:

```
>>> L1 = [1,2,3]
>>> L2 = [2,4,6]
>>> for i,j in itertools.izip(L1,L2):
...     print i,j
...
1 2
2 4
3 6
```

re: Re allows regular expressions to be written into your Python file.

re.search(): re.search() searches a defined set of text for a defined pattern. For example:

```
re.search('Bear', animals.txt)
```

will search our animals.txt file for instances of the word bear. This type of regular expression could be integrated into a loop to rapidly find bits of important text.

A note on hard-coding

On one hand, scripts should be black boxes. They should perform a task without the user knowing how it was done. This concept is known as 'information hiding,' and it serves to protect the script from *ad hoc* changes by the user which would preclude repeatability. On the other hand, if the script performs too specific of a task and cannot be changed responsibly by the user, the user may be tempted to go in and manually change something. A part of the script that must be changed by editing the script itself is known as ‘hard coding.’ Avoid it as best you can and make your script flexible by adding variables that can take input to change whatever you might want to change without editing the script. These include `sys.argv[]` for scripts, and parameters for functions. Default parameter values are a powerful way to control how functions behave.