Week 5 - Functions and Common 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. *Value* is an object and will therefore have a type.

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
```

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','T','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 imagine you wanted to solve the problem from Week 4. It seems overwhelming at first, so you write the following notes to yourself:

1.) Open file and make a list of the contents of each line
2.) Loop through lines and add up observations for each animal
3.) Print observations

You think about how you're actually going to do this a little more, and amend your notes thusly:

1.) Open file and make a list of the contents of each line – strip '\n's and split each line on '\t'. That way each element of the list is itself a list with three objects corresponding to org., site, and count.
2.) Loop through lines and add up observations for each animal – Datastructure? Hmm, I want to make sure I keep my counts for each organism separate. Dictionary! If organism is not in dictionary, add it as key with count as value, if it is in there, add count to current value in dictionary.
3.) Print observations – for each organism in dictionary, print organism and count.

Now look at obs_counter2.py in this week's materials. Functions map nicely to notes! As a bonus, you never have to re-write, say, "opener()." Just cut and paste it into a new script if you want to do something else with it (and when would you *not* need a function like opener()?)

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
```

Common Modules

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.

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.

csv: Comma-separated value files are very common for importing and exporting data from databases. This module will handle a lot of the parsing of datafiles for you. Very handy!

csv.reader(filename, dialect): This will read a given file as a csv, eliminating the need to split lines at the comma. The dialect can be set to different softwares, most commonly excel. For example:

```
with open(file.csv, dialect=excel) as csv:
```

will open the file as a csv and immediately parse any line endings or unusual characters associated with Excel's proprietary format.

csv.writer(filename.csv): This function will write data as a csv file. This handles any parsing into the csv format. We will tlak about some limited databasing later in this course, and the type of file is very useful for common databases such as SQL.

Interacting with your code : functions, sys, and raw_input().

General Form of raw_input():

```
raw_input("stdout string")
```

Prints "stdout string" to the screen and returns a string of whatever the user typed into stdin.

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 by making your script flexible.

Look at obs_counter.py. On lines 43-45, the elements of the lists of observations are counted up. We knew in advance that there would be two observations for each organism, so we added *list*[0] to *list*[1]. But what if you want to run the script on a file with more than two observations? This script will only count the first two, so you'd have to go in and edit it to add up however many observations there were in your new file. That's bad.

The function obs_dictionary() in obs_counter2.py solves this problem with a loop and if/else program flow. If the organism is already in the dictionary, the function adds the observations from that line of the file to the total count already in the dictionary as a value. This type of function, where you want one variable that changes (the running count) and one variable that does not and stays bound to the former (organism name, in this case), is a very common use of dictionaries.

But the function obs_dictionary() will go until it reaches the end of the file and there's currently no way to stop it or to, say, choose which organisms it counts or which field sites it counts from. The script obs_counter3.py illustrates how you might write a script that could take the user's input into account.

There are therefore at least three general ways you could think about avoiding hard coding, none of which are mutually exclusive:

1.) Replace calculations and manipulations with generalizable functions.

2.) Use input from a file (via the `sys.argv` list, see above) to direct how the script works. obs_dictionary() wouldn't be much use without the variable 'infile,' for instance.

3.) Use raw_input() to direct how the script works.