Introduction to Bioinformatics – AS 250.265
Functions and Modules in Python

In this assignment, you will implement a set of functions that perform basic manipulations on DNA (and RNA) sequences. These several functions, which we will term DNATools, will be packaged into a Python module. As a Python module, you will be able to reuse these simple functions over and over again throughout the course in other programming assignments. Additionally, understanding how these functions work will give you a better understanding about how tools like BioWorkbench are implemented.

Many of you have commented that the Python Tutorial is not the most intuitive document to understand, particularly from a non-programming perspective. Thus, the first part of this document will outline some of the fundamental Python concepts we will need to learn before implementing DNATools. As an example, we will use a toy program I wrote called STATTools, which implements some simple statistics functions. You are encouraged to examine the code closely for STATTools, as much of the syntax and especially the usage will be similar between the two libraries.

An Introduction to Structured Programming

Most of the programming we have done in the class so far as been interpreted one line at a time. Predictably, it runs from start to finish, and the only thing that would prevent your program from running each line exactly in order would possibly be a loop. This style of programming is fine, and in fact this is how the original computers were programmed. While Python wasn't the language used, each instruction was fed into a machine via a punch card (or by a series of switches) and the computer operated one instruction at a time without much deviation from the original punch card order.

Then, two things happened. First, people realized that many problems could be solved more easily with a computer's speed and accuracy. Thus, the demand for programmers increased dramatically. Second, as demand increased, it was soon learned that the number of programmers could not match the demand. What was needed was a means of optimizing the programming process: if programming a computer became easier, then the limited number of programmers may be able to match the demand placed upon them more easily.

This demand-supply problem resulted in the development of structured programming languages. Rather than writing programs one line at a time, programmers could take advantage of modularity to reuse their code more easily. As the demand for programming solutions increased even more, steps were taken to make programming conceptually simpler—object-oriented programming languages like C++ (and eventually Java) became popular. The ever-increasing abilities of the machines themselves allowed difficult optimizations to be made analytically rather than relying on the skill of a programmer. As a result, programming is now taught in many high schools, and just about anyone with an interest in learning to program to write their own code while understanding very little about how that code is translated into the actual language of their machine. This is not a bad thing, since presently many interesting problems may be solved without needing to understand the details of how the machine works.

The topics we will learn in this assignment—functions and modules—are nothing more than products of the structured programming revolution. By allowing programmers to group related and repeated statements into *functions* (or subroutines), Python relieves them from having to enter the same code again and again when they want to perform a task frequently. Similarly, by allowing functions to be grouped into *modules* (or libraries), common pieces of code may be shared by many different programmers or across many different programs without making the programmer rewrite this code every time he or she wishes to use it.

Consider the following example: You are working for Microsoft, and Bill Gates asks you to write a program that will read and play MP3's. Meanwhile, he asks your best friend to write a program that will convert an MP3 recording into the lyrics being sung or spoken. Both of these programs will be combined in a finished product used for medical dictation. It's clear that both you and your friend are going to have to have code in common that can read an MP3 file. It would be nice if your friend could reuse your MP3 code in her program. It would be even nicer (for Bill, anyway) if you could both start working on your code at the same time. Functions and modules allow this to happen: at the start of the project, you tell your friend exactly what your code is going to do—what will it take as input, and what will it give as output. Your friend can then continue working on his project, making the assumption that your code will do what you said it will. She doesn't care how you implement the code, as long as it conforms to the specifications you gave her at the start of the project. Thus, you can both work toward the end of the project at the same time. This is called programming by contract, because you and your friend have made a contract about what exactly your programs will do without specifying how exactly they will do them.

In this assignment, you can't work in teams. However, a little foresight shows how code packaging may be useful even for an individual programmer. For one, it allows you to reuse your code very easily: you get the same reuse advantage as above, even if you can't work on two projects at once. Secondly, it provides an easy means for error control and correction. By grouping common tasks in to functions, bugs can be fixed in one place (the function) rather than every place the task would be performed. Grouping code into concise units also helps you to find bugs more easily, as each unit of code performs one conceptual operation that can be understood more fully than the sum of its individual instructions.

With that introduction, let's look at functions and modules, the two means that Python provides for structuring your programs.

Functions in Python

Recall that Python evaluates statements one chunk at a time. Given the expression below, Python must first simplify the right hand side of the equation before assigning a final value to the variable a.

```
a = (1. + 4.)/(3. * (4. + 2.))
```
→ First, evaluate the innermost addition in parenthesis and replace it with the result
```
a = (1. + 4.)/(3. * 6.)
```
→ Next, evaluate the sum and replace it with the result
```
a = (5.)/(3. * 6.)
```
→ Next, evaluate the product and replace it with the result
```
a = (5.)/(18.)
```

→ Next, evaluate the division

```
a = 0.2777777777
```

→ Finally, assign the result to the variable a

Now, suppose we needed to calculate factorials repeatedly (perhaps we're taking a probability course). Specifically, suppose we wanted to calculate the number of ways to choose 4 items from a bag of 12 (4-choose-12), where the order of items is irrelevant. Probability states that this is given by:

$$N = \frac{12!}{4! \cdot 8!}$$

One way to solve this in Python is to write the following program:

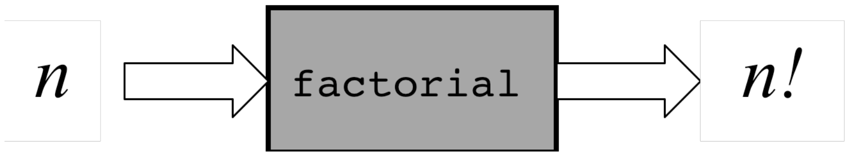
```
i=12                                # Calculate first factorial
fac_12 = 1
while i > 1:
   fac_12 = fac_12*i
   i = i – 1

i=4                                 # Calculate second factorial
fac_8 = 1
while i > 1:
   fac_12 = fac_12*i
   i = i – 1

i=8                                 # Calculate third factorial
fac_8 = 1
while i > 1:
   fac_12 = fac_12*i
   i = i – 1

print fac_12 / fac_4 / fac_8        # Print the result
```

On the other hand, it would be much more convenient if we could package the code used to calculate the factorial into one unit. Then we wouldn't have to duplicate it over and over again whenever we need to calculate a factorial. Conceptually, we want a "black box" that takes an input a number and returns as output the value of the factorial of that number, like so.



In this example, the *n* on the left is called an argument, and the result of the function would be its *return value*.

If such a function were defined, then we could perhaps make a statement like this:

```
print factorial(12) / factorial(4) / factorial(8)
```
→ Evaluate the first factorial result
```
print 479001600 / factorial(4) / factorial(8)
```
→ Evaluate the second factorial result
```
print 479001600 / 24 / factorial(8)
```
→ Evaluate the third factorial result
```
print 479001600 / 24 / 40320
```
→ Next, evaluate the first division
```
print 19958400 / 40320
```
→ Evaluate the second division
```
print 495
```
→ Finally, print the result to the screen

Now that we've established the concept of a function, and specifically how Python evaluates functions, we need to determine the syntax involved in defining a function. As with most Python constructs, you can bet that indentation will define part of the function and that a colon will be involved.

The Syntax of Python Function Definitions

The Python function definition for factorial is given below. To make things convenient, I've appended line numbers. We'll reference them in our discussion of how to define a function in Python.

```
def factorial(n):                        # Line 1
   i = n                                 #      2
   result = 1                            #      3

   while i > 1:                          #      4
         result = result * i             #      5
         i = i – 1                       #      6

   return result                         #      7
```

The first part of a function definition begins with the `def` keyword. When Python sees this keyword, it expects that the next token it reads will be the name of the new function you are defining, in this case "factorial." After it sees the name of the function, it looks for the argument list, offset by parenthesis. Since factorial will only take one argument, we have simply put n between the parenthesis. This tells the function that, in the body of the function, we will name (dub) our argument *n*. (Note that we reference n in line 2, so this variable becomes defined for the scope of the function.) The rest of the code in lines 3 through 6 perform the actual work of the function. Once the work is completed, we must tell Python how to evaluate the function. When Python reaches the return statement on line 7, it understands that the function has finished, and it uses the expression following the return statement to evaluate the function. Any indented code beyond the return statement (not shown here) is ignored—the function "exits" when it reaches the return statement.

Another example may be helpful. Consider this (rather absurd) renaming of the multiplication operator.

```
def multiply(x, y):                        # Line 1
   result = x * y                          #      2

   return result                           #      3
```

Here we see a variation: we've increased the number of arguments to our function. Indeed, the number of arguments can be arbitrarily long, as long as each value is separated by commas. If we were to call `multiply(3, 6)`, x would be assigned the value 3, y would be assigned the value 6, and the result would be 18.

Let's return now to the example of calculating the combinatorics problem. Below is what our final program would look like to calculate 12-choose-4. Clearly this is much shorter, and the final line is much easier to conceptualize.

```
def factorial(n):                      # Function definition for
   i = n                               # factorial calculations
   result = 1

   while i > 1:
        result = result * i
        i = i – 1

   return result

print factorial(12) / factorial(4) / factorial(8)
```

Now, when `factorial(12)` is evaluated, Python assigns n in the function itself the value of 12. Then, it completes the execution of that function, and returns the result of 12!. The next time the function is called, n is assigned to be 4, and the value of 4! is returned. Finally, n is assigned to be 8, and 8! is returned, and the division proceeds as before. The code in the factorial function is reused by Python, so you don't have to type it over and over again.

In your assignment, the templates for the functions are written out for you, and you won't actually have to bother with the definitions yet. However, this section should provide you will all you need to know to start writing your own functions to organize your code.

Python Modules

Now that you've written several functions, the next step is to increase the modularity of your code even further. This is done through the *module* system of Python. Fortunately, there isn't any new syntax to learn in *creating* modules—by understanding functions, you know it already! The only syntax you will have to learn is how to *use* modules.

The module system of Python simply allows code written in one Python file to be reused in another program file. For example, suppose I have implemented the following functions in a program called `algebra.py`: `quadratic_eqn`, `roots`, and `linear_solve`. One might imagine that these functions solve the quadratic equation, return the number of real roots, or solve a linear system of equations, respectively. Now suppose that in another program `solver.py`, I would like to use those functions without having to copy and paste them again. To do this in Python, I would utilize the *import* command.

```
import algebra
```

Once Python reaches this command, all of the functions in `algebra.py` would become available to `solver.py`, *provided `algebra.py` is in the same folder (directory) as `solver.py`*. There is only one catch: in `solver.py`, the functions I imported would have to be prefixed by the name of the algebra module first. The complete code for `solver.py` could look something like this:

```
import algebra

a = 16.
b = 4.
c = 3.

print algebra.quadratic_eqn(a, b, c)

print algebra.roots("5x^3 – 4x^2 + 3x – 8")

a1 = 4
b1 = 3
c1 = 0

a2 =  4
b2 = -3
c2 =  8

# Equation is of the format:
# a1*x + b1*y = c1
# a2*x + b2*y = c2

print algebra.linear_solve(a1, b1, c1, a2, b2, c2)
```

The code for all three functions is now accessible, but it must be referenced based on the name of the module (i.e., the file name without the .py extension), and a period must follow the module and precede the function name as shown above.

Python uses modules extensively to subdivide and organize its built-in functionality. As you may have already noticed, there is a math module (`import math`) that contains all of the trigonometric and logarithmic functions (`math.sin`, `math.tan`, `math.log`). There are many other modules, too, and you may browse them on the Python documentation website.

STATTools: An example of a Statistical Function Library

On the assignment 2 website you will find a pair of files called `stattools.py` and `stattest.py`. Download these two files and save them to the same directory. Then try running the `stattest.py` program. It displays a list of integers and displays some standard statistical analysis on the list. As you look at the code at this point, you should be able to understand how the test program is able to access functions in the tools program, even if you can't understand how every function works. Experiment with these files. When you become comfortable with functions and module referencing, you will be ready to start the assignment.