*An introduction to Python 2.7 and Anaconda*

*Contents*

1. *Installing Anaconda*
2. *How to run Python scripts*
3. *Importing modules*
4. *Reading in data files to Python*
5. *Output to screen or file*
6. *Mathematical operations*
7. *Object types*
8. *Logical constructs*
9. *Dictionaries (hashes)*
10. *A detailed example*
11. *Installing Anaconda*

There are a lot of different options available for Python, but Anaconda is the best for scientific applications. Anaconda contains Python 2.7 (get it? Anacondas eat other snakes…) as well as a host of commonly used modules like Numpy (number python, a matrix algebra suite) and SciPy (science python, which includes lots of data handling tools). Anaconda is also easy to install, as it automatically does all of the work when it comes time to modify the default file path of your computer so that you can run Python from any terminal. To download Anaconda, just Google it. You can get the full version for free from its owner, Continuum Analytics.

1. *How to run Python scripts*

The install of Anaconda will include several different programs which will allow you to run Python in different ways. The QTConsole will allow you to run but not save scripts, so if you want to use it you will need a text editing program like Programmer’s Notepad. Your best bet is Spyder, which is an interactive development environment (IDE). Spyder does a host of useful things for you, like keeping track of all of your variable’s types and their current values. It is also helpful for bug-checking because it will show you likely problems before you run the script.

1. *Importing modules*

The base functionality of Python is actually quite limited, and so you will often find that in order to do anything useful you need some added stuff. Anaconda brings in a lot of useful modules, and the most useful one, SciPy, is automatically called with every script, so you don’t have to manually load it. There are three others which you will probably use often: numpy, matplotlib, and sys. Numpy is a matrix algebra suite, matplotlib is a graphing tool, and sys is a process control suite which allows you to make in-line error checks (the type of checks which matter to us but not Python, like making sure probability is conserved). Anytime you need to import these or other modules, you call them at the top of your script. Ex. To import numpy, write “from numpy import\*” or, alternatively, for sys, “import sys”. The first way allows you to use numpy functionality without calling it specifically as numpy.function(), while the import method for sys will require that you use “sys.exit()” rather than “exit()”. If that’s confusing, don’t worry, there is an interactive example later.

1. *Reading in data files to Python*

As computational chemists, we typically work with large data sets, often in the form of giant columns of numbers and/or biological sequence information. Python includes several different ways to read-in these types of files so that they can be manipulated. The most useful to you will probably be loadtxt and filehandle. loadtxt is part of numpy, while filehandle is a basic Python function. Here is an example of loadtxt:

k\_a = loadtxt("C:/Users/dan182/Documents/Python Scripts/Inputs/k\_a\_3.2.txt")

This line loads in the file “k\_a\_3.2.txt” from my documents folded and saves it as an array with the name k\_a. Note that / rather than \ is used in the file path. Python does not recognize \ as a character and will give an error, despite the fact that Widows uses \ to define file paths. Here is an example of filehandle:

fileHandle = open ( 'C:/Users/dan182/Documents/Python Scripts/Inputs/SFVP\_codon\_sequence.txt', 'r' ) #open as read only the file containing the codon sequence for SFVP

The # indicates that everything which follows to the right of it is a comment, and is not compiled. fileHandle is opening, but not saving, this codon sequence file. Other commands are needed in order to manipulate the contents:

rawtext = fileHandle.read() #let rawtext hold the values found in the file

fileHandle.close() #close the file (but rawtext still holds the values!)

Here I define the temporary variable rawtext to hold the contents of the text file, and then close the file. It is considered best practice to open, copy, and then close files rather than directly modifying their contents, as this can lead to some serious issues down the line.

So, why are there two different file input functions? Loadtxt is numpy functionality, and so it automatically brings in the file contents in a numpy.array, a specific data structure which is not optimal for handling strings (see the section on types).

1. *Output to screen or file*

Once you have written some code and done something useful, it is nice to be able to save and store the output. Alternatively, you can print it out on the window.

Printing to screen is easy, requiring the simple command “print x” where x is the variable. If we want to print a series of variables they must be separated by commas: “print x, y, z”. You can print any string of characters by putting them in single quotes: “print ‘some stuff’”. If you have an element A and you want to call the 6th element, you would write “A[5]”. The reason why it reads 5 and not 6 is discussed later, in the logical construct section.

Output to file requires the command savetxt, which is part of numpy. The following lines of code allow the script to take user input for the filename and then save it to a directory in my documents folder:

y = raw\_input("Enter a name for the output file: ") #take user input for file name

savetxt("C:/Users/dan182/Documents/Python Scripts/Outputs/SFVP\_" + y +".txt", hstack([t\_matrix,((t\_matrix\*dt)-incorporation\_period), P\_matrix,]), fmt=['%g','%.10f','%.10f'], delimiter='\t')

The savetxt command is used here to print out three columns of data, each with their own format, separated by tabs. This gets a little complicated, so for now let’s just say that you can Google the format if you need to.

1. *Mathematical operations*

Multiplication: use \* to multiply

Addition: +

Subtraction: -

Division: /

Sum(A[x:y]): sum the terms of the array A from index x to y

Python follows normal mathematical order of operations.

1. *Object types*

All programming languages have different data types which determine how they are treated in functions. Some of them will be familiar to you from mathematics, but some of them will likely seem strange or be handled differently than you are used to.

int: ints are integers, and have some unique uses. Only integers can be used to set the bounds of loops (discussed later). ints are potentially problematic in calculations; Python does floor/ceiling division with integers, meaning that any and all remainder is thrown away. So, for example, as far as Python is concerned, 2-1.2 = 0. This can seriously break some code if you are not careful.

dec : decs are decimals, and any number can be initiated as such by adding a decimal point after it. decs are useful but should not be used for complex mathematics, as repeated operations result in rounding errors due to the limited precision of dec types. Numpy floats, especially 64-bit numpy floats, are a better choice.

float: a floating point number. This should be your default type for mathematical operations. Numpy allows you to use either 32- or 64-bit floats, and you should probably always use 64-bit floats to ensure precision.

Str: str is short for string, and means any group of alphabetical or numerical characters. Any number, even a decimal, can actually be made into a str if you so desire, and there are some cases where this is useful.

The Spyder IDE will keep track of all types for you, but if you ever want to explicitly check the type of a variable x, the command “type(x)” will return the type. You can change the type of an object, within certain limits, using the commands int(), dec(), float(), and str(), where the variable you are converting is placed within the parentheses.

1. *Logical constructs*

Computer code relies on logical constructs to order work flow. The most important are conditional statements and loops.

Conditional statements in Python consist of “if”, “elif”, and “else”. There can only be elif or else conditionals if there is an if. For example:

if x = 2:

Print “x is 2, but who cares”

elif x > 2:

Print “x > 2, still don’t care”

else:

print “x < 2, really don’t care”

There are two types of loops in Python (and most other languages), for and while loops. For loops do something for a certain numbers of iterations. While loops do something until a given condition is no longer true. Examples follow:

for x in range (0, 20, 1): # format: (start, stop, increment)

print x

while x < 20: # while x is less than 20, print x and increment by 1 each loop

print x

x = x+1 # if we don’t modify x then this loop prints forever!

You may have noticed that the for loop will only print 0 through 19. This is because loops in Python are inclusive for the first number and exclusive for the final number. This is actually advantageous to us. If an array in Python has 10 elements, then they are called as elements 0 through 9 rather than 1 through 10, meaning that you can loop over an array from 0 to 10 and precisely fill each element of the array. On a side note, the final element of an array is called as the -1 element.

1. *Dictionaries (hashes)*

Dictionaries, sometimes called hashes, are constructs which relate two objects , regardless of type, to one another. You can make an empty dictionary by choosing a name, like test\_dict, and writing:

test\_dict = {}

In order to fill the hash you need to use the following commands:

Test\_dict[1] = number\_one

So, if we call test\_dict[1] we should see “number\_one” printed to the screen.

1. *A detailed example*

With this document you should have gotten a package of input files which represent the mRNA nucleotide sequence for Semliki Forest Virus Protein (SFVP, a viral polyprotein assembly), the translation elongation rates for each codon, and the corresponding list of codons.

A quick suggestion: variable names need not be short, as this is not mathematics. My first scripts had variables with names like “a” and “x”, but it is very useful to use descriptive names that allow people to tell at a glance what an object is. For example, don’t call your calculated probability “P”, call it “probability\_of\_folding\_at\_equilibrium”. Annoying to type? Absolutely. Easier to use and make sense of your code? Oh my, yes.

With that said, try to write a script with the following functionality:

1. Read in the input files. They are all text files, but be careful how you import them!
2. Create a hash which will relate a codon to its mean translation rate
3. Parse the SFVP sequence information into codons, which are strings of three codons
4. Create another hash and use it to associate the correct translation elongation rates with each codon in the SFVP transcript
5. Print the codon/translation rate pairs both to screen and to file

This is actually extremely useful information, and is the first step in modeling the codon translation rate effects on a co-translational folding curve for SFVP. SFVP is a polyprotein assembly derived directly from the sense-RNA genome of SFV, and it is able to fold co-translationally, allowing the virus to quickly and efficiently self-process its proteins. Other much more dangerous viruses, like poliovirus, use the same expression strategy. The influence of variable codon translation rates on nascent protein behavior are on the cutting edge of modern protein science, and are a specialty of this group!