A brief tutorial for Python scripting

Practical Course in Bioinformatics (Bio 334)

By Rzgar Hosseini

Section – Andreas Wagner

Note: The folder containing this document contains other text files called glycolysis, sequence data and fasta_sequence.txt. In addition, it also contains the metabolic network of E. coli - iAF1260.xml, the glucose environment file glucose.flx and another directory titled read_directory. All of these files and folders will be used below to help you learn Python.
Python is a widely used general-purpose, high-level programming language. Its design emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. Python interpreters are available for installation on many operating systems, allowing Python code execution on a majority of systems.
Many programming languages, such as C, need to be compiled before execution. In contrast,Python uses an interpreter, which means that your computer interprets (and executes) each line or block of lines in the Python program as it scans that line. You need few syntactic elements to write simple Python programs; hence Python is a highly popular language with biologists. This simplicity makes Python especially suitable for writing programs for one-time use. In this Python training module, we will go over some fundamental concepts of Python. More focus is given on writing functional scripts rather than making you an expert at Python.

1. The simplest Python script

A Python script begins by defining the location of the Python interpreter, beginning with the characters #!
The interpreter interprets your program, converts it into machine-readable code and then executes it. Open a text-editor, paste the following lines, and save them as the file script.py.

```
#!/usr/bin/python
print "This is easy!\n"
```

To run it on a system that uses a variant of the Unix operating system, you may need to change the permissions of script.py so that you can execute it. At the shell prompt, do the following -

```
chmod a+x script.py
```

[press enter – your script is now executable]

Type the following at the shell prompt

```
$ python script.py
```

You should see “This is easy” printed on your screen with the shell prompt on the next line.

The “\n” indicates the “newline” character. Without this character, Python doesn't skip to a new line of text. If you were to type only the letter “n” instead of “\n”, it would mean the character n. To “escape” its normal meaning, we need to tell the Python interpreter to interpret it in a specific manner by using the backslash character '\'. (What happens if you skip the “\n“?)

2. Python object types

Variables correspond to memory locations with specific names that can hold information. This information defines the type of the variable in Python. Python has several types of variables – numbers, strings, lists, dictionaries, tuples, and files. Of these, we will mostly use numbers, strings and lists. For example, we can assign a string to variable name and a number to the variable age as follows:

```
name = "Vardan"
age = 24
```

NOTE. Python is case-sensitive (`age` is different from `Age`)

Python can also insert the value of a variable into another statement, as in the following example.

```
print "His name is %s and his age is %d years.\n"%(name,age)
```

Note that %s is used for inserting the value of a string and %d is used to insert the value of an integer into the above statement. To insert the value of a variable into the statement % followed by the variable name is written after the statement. If we insert two or more variables we put the variable names in a parenthesis %(variable_name1, variable_name2,etc.) as we used in the example above.

The above Python line will yield the following output.

```
His name is Vardan and his age is 24 years.
```

Numeric variables contain numbers and can be used for arithmetic operations. Here is an example.

```
a = 5
b = a*5         # the value of b is now 25
b *=10               # the value of b is now 250. b (25)
multiplied by 10
```

The character “#” is used to insert comments into code. It tells the interpreter to disregard all characters that follow it and that are on the same line. It is good practice to comment your code with descriptions of what you are doing, so other people can understand your code, especially when working in teams. In fact, you should always comment your code. It’s almost a given that you won’t be able to understand your own code after a while, if it’s uncommented.Note that you don’t need to comment each line of your code, comments should explain the logic behind a particular block of code.

3. Lists

Lists are sequences of elements indexed by an integer. The elements of a list can be as simple as numbers or strings, but they can also be lists, tuples of dictionaries, More importantly, lists can contain elements of mixed types. For example:

```
nums = [1, 22, 543, 1024]
strs=["Cat", "Dog", "Horse", "Dinosaur"]
list_of_lists = [ nums, strs]
```

This command specifies a list called nums, which contains four numeric variables. The indexing of a list starts from zero. So the third scalar in nums is nums[2], which has the value 543. Note that the last element of this array is nums[3]. nums[4] does not exist. It is a very common mistake to try to index array elements that have not been specified, and can have fatal consequences in Python programs. To print the third element of the list, we could use the command `print nums[2]`

which would yield the output

```
543
```

Write a program that prints all list elements, together with the index of each list element.

You can find the size of a list by using the list name in the function **len**. (Similar to its usage in mathematics, a Python function performs an operation on some input or argument, in this case the list nums, producing some output, in this case the length of the list.)

```
size_length = len(nums)
print "%d\n" %size_length
```

which assigns the number of elements to the variable size_length.

4. Some string manipulations

In Python, you can easily split a string or any data into separate parts using the built-in **split** function. You can store the parts in an array and access them as needed. The program

```
str1 = "I am a student"
fields = str1.split(' ')      #results in fields=['I', 'am',
'a', 'student']
type(str1)
type(fields)
```

Here is what the second line of the program does. The words in the scalar variable str1 are separated by a single white space character. The function split is used to split str1 wherever a white space occurs in it. The character(s) at which the function splits the string is given as the argument between the two single quotes. (In this case, the character is a blank or a so-called *white space*, but any character could occur between the two quotes.) (Try replacing the second line in the above program with fields = str1.split ('a'). What happens?)

The part (substrings) into which split divided the string are assigned to the list called fields in the order they occur in the string. Instead of specifying a white space, you can also use any arbitrary character (i.e. delimiter) to split a string. For example:

```
str2 = "This_is_very_easy!"
fields = str2.split('_')     #results in fields=['This', 'is',
'very', 'easy!']
```

To compare two strings in Python, we can easly use the logic operator "==". For example:

```
String1="horse"
String2="house"
String3="horse"
```

```
>>>String1==String2
False
>>>String1==String3
True
```

Sometimes whitespace characters that exist by default at the end of strings cause string comparison unreliable. To get rid of whitespace characters at the end of strings we use **rstrip** function. The syntax is as follows:

str.rstrip()

This strips whitespace characters from the end of the string named str.

Another important string manipulation that Python supports is string concatenation which is simply done by summing two strings. For example:

```
Str1="Re"
Str2="combination"
Str=Str1+Str2
```

Results in:

```
Str="Recombination"
```

5. if and the elif statement

A control structure allows the evaluation of commands only if certain conditions are met. A simple control structure such as the "if" statement has the form –

```
if  expression:
      statements
```

This means that **if the expression is true**, then execute one or more commands. For example

```
if  i < 10:
       j = 100
```

The above program checks if the value of the variable i is less than 10. If it is, then the variable j is assigned the value 100. Note that after the expression to be evaluated, the colon **":"** is necessary. Also note that in Python, unlike in some other programming languages, the statement (or block of statements) to be executed after the "if" is not written in curly brackets { }. Instead we use indentation to distinguish them and the indentation is necessary, otherwise an error in the program's execution ensues. Even worse, wrong indentation can lead to logical errors that will not be detected by the interpreter. This will lead to unexpected behavior (bugs) in your code.

Another closely related control structure is the if-else statement, which allows for the possibility that an expression may not be true, and executes one or more commands in that case.

```
if  i < 10:
        j = 100
else :
        j = 0
```

If i is less than 10, then j is assigned a value of 100. Otherwise, that is, if i is equal to or greater than 10, j is assigned a value of zero. **Important**: To check if a variable has a specific numeric value, you should use == as in this case

```
if  i == 10:
    j = 100
```

(The == operator indicates comparison, whereas plain = indicates assignment of a value to a variable.)

The program above tests for equality between the scalar i and the number 10. We can write a sequence of if and else if statements as follows:

```
if  i < 10:
       j = 100
elif i==10 :
       j = 50
else:
       j=0
```

The if statement below checks if i is less than or equal to 10.

```
if  i<=10:
     ….
```

(To perform a greater than comparisons, use >=)

In many situations one needs to compare strings, and especially DNA strings. String comparison uses the same operators as comparison of numbers in Python. For example, the following program checks if two patterns of sequence are the same,

```
pattern1 = "TATA";
pattern2 = "TGTG";
if pattern1==pattern2:
     print "Found the regulatory TATA box\n"
else:
        print "No TATA box found\n"
```

6. The if not statement

The if not statement has the form

```
If not expression:
          statements
```

This means that **if the expression is false**, then execute one or more commands that follow. For example,

```
i = 0
j = 0
if not i > 0:
        j = 100
```

In this case j is assigned a value of 100 because the expression i > 0 is evaluated as false. You can view the if not statement as the opposite of an if statement. The if statement executes commands if the expression in it is true, while the if not statement executes commands when the expression is false. (Try changing the expression in the above code snippet to i == 0 and check the output.)

7. The for loop

The for loop has the form:

```
for <target> in <object>:
       <statements>
```

The following is an example:

```
numbers = [5, 10, 15, 20]
for element in numbers:
      added = element+1
             print "%d\n" %added
```

The for loop in this program moves through the list (numbers) and at each step assigns an element of the list (numbers) to variable element. It increases the value of element by 1 and stores it in variable ”added” and finally prints the variable “added”.

Python has a useful function called range that is used with for loops. It has the following syntax:

```
 range(starting_number,ending_number)
```

It receives two arguments starting_number and ending_number and outputs a sequence of integers starting from the first argument (starting_number) and ending at the second argument (ending_number).

For example:

```
subject=["Biology","Biophysics","Biotechnology","Bioinformatic
s","Bistatistics"]
for i in range(0,4):
    Print "He studies %s\n" %jobs[i]
The output is:
He studies Biology
He studies Biophysics
He studies Biotechnology
He studies Bioinformatics
He studies Biostatistics
```

8. The while loop

The while loop has the form:

```
while <test>:
       <statements>
```

As an example:

```
x = 0
while x < 5:
          print "%d\n" %x
         x=x+1
```

The statement “while (x<5):” functions like an if statement which continually checks if the value of x is less than 5. The second command in the indented block increments the value of x by one (x=x+1). The while loop terminates once the value of x is equal to 5. Remember that the statements in the block have to change the value of the variable that is being used in the condition. (What would happen if x+=1 is removed from the indented block above?) . Recall that indentation defines blocks of code that are executed together. In this case indentation defines the code that is the part of the while loop. How does the example below differ from the one above?

```
x = 0
while x < 5:
     print "%d\n" %x
x=x+1
```

9. Continue and break

The continue statement causes the current loop iteration to skip to the next value or the next evaluation of the control statement. It is best explained with an example

```
array = [1, 4, 3, 4, 2]
for number in array:
         if number == 3:
          continue
         else:
                 print “%d\n“ %number
```

The output is

```
1
4
4
2
```

The program skips the print statement and iterates to the next value in the list if the variable “number” is equal to 3.

The break statement terminates the current loop. For example:

```
array = [1, 4, 3, 4, 2]
for number in array:
         if number == 3:
          break
         else:
                 print "%d\n" %number
```

The output is

```
1
4
```

The program exits the while loop when the number is equal to 3.

10. The append() method

The append method allows you to append a list, that is, add elements at the end of the list. This function can be helpful when you need to compile certain values and analyze them later in your program. The append function has the form:

```
ListName.append(value)
```

For instance, the list called a_list currently contains 4 numbers:

```
a_list=[1,2,3,4]

a_list.append(10) will add 10 to the end of the list, so the
new list will be:

a_list=[1,2,3,4,10]
```

11. Reading a file into a Python program

Python can also be used for text parsing, and bioinformaticians often use Python programs to analyze large amounts of data contained in a text file, such as a file containing many DNA sequences. The data in such a file needs to be read into the Python program. There are several ways of parsing a file, but the basic construct remains the same. In the folder called Python_trainer, there is a file called glycolysis.txt. It contains the reactions involved in the glycolysis pathway. An example line from the file is as follows

PFK: f6p + atp --> fdp + adp
PGI: g6p <==> f6p

Here, PFK is the name of a chemical reaction, and the colon (:) delimits this name from the reaction equation. The --> represents an irreversible reaction and <==> represents a reversible reaction.

```
myfile = open('glycolysis,txt','r')
```

The command above will associate the filehandle myfile with the name glycolysis.txt. Filehandles are used to refer to a file, read from it, or write to it. The second argument, ‘r’ designates the mode of opening the file. In this case the file is opened for reading. Opening the file for writing will be discussed in the next section. After opening the file, the file can be read by three alternative methods. For the example above, we can read the opened file as follows:

myfile.read() read entire file into a string.
myfile.readline() read the next line of the file through end-line marker
myfile.readlines() read the entire file into list strings, one string per line (this is not recommended for large files)

The following is a simple example demonstrating how to read a file and print each line in the file.

```
#!/usr/bin/python

myfile=open('glycolysis,txt','r')
S=myfile.readlines()
for line in S:
      print "%s\n" %line
myfile.close()
```

The output should be all the lines in the file glycolysis.txt printed on your screen. The statement myfile.close() closes the open file. This is important especially for large files, in order to use the computer’s memory efficiently
Sometimes a program needs multiple files as input. This can be accomplished by using open function multiple times, and by assigning each file a distinct file handle. For example:

```
file1=open('file_name1.txt','r')
file2=open('file_name2.txt', 'r')
do something with either file…
file1.close()
file2.close()
```

In the glycolysis.txt file, each line contains a reaction name and a corresponding reaction equation separated by the : delimiter. That is, reaction names form a first column and reaction equations form a second column of the file. One can use the delimiter to print only one of the columns, using the split function explained in Section 4 above.

```
#!/usr/bin/python

myfile=open('glycolysis,txt', 'r')
```

```
S=myfile.readlines()
for line in S:
      fields = line.split(':')
      print "%s\n" %fields[0]
myfile.close()
```

The statement fields = line.split(':') splits each line based on the : delimiter into sections (two sections in our case) delimited by the colon, and assigns them to the list fields. The output of the program consists of all the reaction names in the file.

12. Writing to a file

In the preceding sections, we learnt how to read from a file. Often, bioinformaticians read from a file, process the data in a Python program and are then required to store their output in another file. We thus need to know how to write into a file. The syntax to do so is

```
myfile=open("filename.txt", 'w')
myfile.write("write something\n")
myfile.close()
```

The variable myfile in the first line is the file handle, as explained earlier. The first line opens a file called filename.txt, and myfile.write() prints the string contained in parentheses to the file. The close statement then closes the file. . Forgetting to close the file you have just written to can cause you to lose some of the program's output.

Opening "filename.txt" with the "w" parameter overwrites the content of myfile.txt, if the file already exists in the current working directory. To avoid overwriting, one can pass "a" (for "append") instead of "w". This enables the program to append to the file. For instance, execute the following program:

```
#!/usr/bin/Python

myfile=open("filename.txt", 'w');
name = "Chuck Norris";
for i in range(1,3):
         myfile.write("%s\n" %name)
myfile.close();
```

Open filename.txt. It should contain the following

```
Chuck Norris
Chuck Norris
Chuck Norris
```

Now replace ”w” with “a” and execute the program again. What do you see?

13. Basic Regular expressions

Python is an excellent language to identify specific patterns in textual data and extract them. You can think of a regular expression as a broadly defined pattern of characters that a string may contain. You have already used a regular expression earlier in the split function, which split a string at all positions where a colon occurred, by using the first argument **’:’** The colon : between single quotes here serves as a regular expression.
The general syntax for using regular expressions in Python is as follows:

```
import re
result=re.search(pattern,string)
```

Note that we have to import library re in order to use the functions related to basic regular expressions. The variable “result” will be TRUE if the pattern is found in the string and FALSE otherwise.

We will use a simple example:

```
import re
if  re.search('World','Hello World'):
         print "It matches\n"
else:
         print "It doesn’t match\n"
```

In the case above, “Hello World” is the string we want to search for the pattern “World”. The string in which we search the regular expression can also be replaced by a variable

```
string = "Hello World"
pattern="World"
if  re.search(pattern,string):
         print "It matches\n"
```

For the different regular expressions given below, report if there would be a match or not?

string = 'Hello World'

```
1. re.search("world",string)
2. re.search("o W",string)
3. re.search("oW",string)
4. re.search("world",string)
```

Answer:

```
1     re.search("world",string)    # Does not match
2     re.search("o W",string)       # Matches
3     re.search("oW",string)        # Does not match
4     re.search("World",string)   # Does not match
```

The first case is not a match because pattern matching is case-sensitive. That is, our string contains “World” and not “world”.

Suppose you had to extract all DNA sequences from a set of sequences that have the pattern “TATA” embedded in them. Use a text editor to open the file sequencedata.txt in the folder Python_trainer. The file contains six short nucleotide sequences. Write a Python program to print a sequence if it contains the pattern TATA.

Answer:

```
myfile=open('sequencedata,txt', 'r')
S=myfile.readlines()
import re
for line in S:
      if re.search("TATA", line):
          print "%s\n" %line
myfile.close()
```

The statement re.search(“TATA”,line) is an evaluation that is true if and only if the string line contains the pattern **TATA**.

We will now go back to our file glycolysis.txt. Write a program to compute the number of reversible and irreversible reactions in glycolysis.txt. Remember, the pattern --> denotes an irreversible reaction and the pattern <==> denotes a reversible reaction.

Answer:

```
#!/usr/bin/Python
myfile=open('glycolysis.txt', 'r')
S=myfile.readlines()
import re
rev=0
```

```
irrev=0
for line in S:
      if re.search("<==>",line):
           rev+=1
      elif re.search("-->",line):
           irrev+=1
      else:
myfile.close()
print "The number of reversible reactions in the model is %d
and the number of irreversible reactions is %d.\n" %(rev,ir-
rev)
```

The program's output is -

The number of reversible reactions in the model is 13 and the number of irreversible reactions is 7.

The program above checks for the pattern <==> representing a reversible reaction. The only other pattern in a line is --> , which represents an irreversible reaction.

Python also has built-in special characters with reserved meanings.

For example, square brackets indicate a set of characters. You can use this fact to search for:

1) sets of digits and characters
   [atg] will match a string if it contains either of a, t, g
2) ranges of digits and characters
   [0-4] will match a string if it contains any of the characters between 0 and 4
   What does [a-z] match?

For example, one can check if a string has any digits in it:

```
string = "Winterthurerstrasse 9"
pattern ='strasse [0-9]'      # [0-9] matches any single digit.
```

The pattern from the example matches if the string contains the pattern "strasse" followed by **any** digit from the designated range, because [0-9] matches all single digit number. That is, it represents a *range* of numbers. (Does the pattern "strasse [0-7]" match this string? Try,)

Just like numbers, one can also specify ranges of characters for matching, by using, for example [a-z]. One can also group specific patterns. For example, if one is searching for addresses on one of the two specific streets Winterthurerstrasse or Bahnhofstrasse

one can use

```
pattern = "(Wintherthurer|Bahnhof)strasse"
```

(The character | designates a logical **OR**.) The regular expression will match if and only if the sub-patterns Wintherthurer or Bahnhof are followed by the pattern strasse, but not other strings, such as the town Winterthur or Bahnhofsquai. To match the pattern Bahnhof followed by any letters, one would use

```
pattern = "Bahnhof([a-z]*)"
```

that is, the sub-pattern Bahnhof followed by any characters ranging from a-z. The asterisk * is a special character, which means that the preceeding character, element or group may occur zero or more times. A plus + means that the preceding character, element or group may occur one or more times, while a question mark ? matches zero or one times.

Here is a summary of these pattern matching rules, plus a few additional ones.

" [0-9]+ "	# matches any sequence of one or more digits
" [a-z]+ "	# matches any sequence of one or more lowercase characters
" [A-Z]+ "	# matches any sequence of one or more uppercase characters
" [^0-9] "	# matches any non-digit character. ^ means negation or not
" [0-9]{5}"	# matches any sequence of 5 digits, specified by the value in {}
" [0-9]{5,10}"	# matches any sequence of 5 to 10 digits
"^[0-9]+"	# Does **not** mean **negation**. It means that the line **starts** with digits
" [0-9]+$"	# Matches a string ending with digits
" [0-9]*|[a-z]*"	# matches any sequence of digits **or** lowercase characters
".+"	# matches to one or more occurrences of any character, extensions of any length.

Sometimes, the pattern you are looking for will contain a character that is a special character, i.e., `-` `.` , *, [, (. To match these characters, you will have to "escape" their special meaning first. You can do this by using the escape character \.

For example, consider a list of files named file.txt, text.tx, lib.h, file

If we want to find only the filenames containing the file extensions, we can use the following regular expression

pattern="\..+"

\.	looks for a dot in a string, because the character \ escapes the default meaning of ., which matches any character
.+	matches to one or more occurrences of any character (this will match extensions of any length)

Similarly, you would use \+ to match "+" or \(\) to match "()".

14. Using regular expressions to manipulate strings

In addition to using regular expressions to find patterns, you can also use them to modify patterns. This makes regular expressions very powerful and flexible search-and-replace tools. The command you need to know is

```
re.sub(pattern,repl,string)
```

the function above returns a modified version of the string after substituting **all** occurrences of the pattern in the string by repl.

Sometimes you just want to replace the first occurrence of the pattern:

re.sub(pattern, repl, string, count = 1)

Here is an example how to replace part of a string:

```
import re
string = "Regular expressions are hard and rocks are also
hard."
new_string=re.sub("hard","useful",string)
print new_string
new_string=re.sub("hard","useful",string, count = 1)
print new_string
```

The output would be

```
Regular expressions are useful and rocks are also useful.
Regular expressions are useful and rocks are also hard.
```

Regular expressions can also be used to split strings, especially when we need complex patterns to split strings. For example if we want to split the string str in the points where “/”or “.o” are placed, we can use the following code:

```
str="/home/rzgar/recombination/results.out.output"
import re
fields=re.split('/|.o',str)
```

results in

```
fields=["home","rzgar","recombination","results","ut","utput"]
```

15. Listing files contained in a directory

An important trick is to know how to list all files in a directory or folder, especially if one wants to analyze all files in a directory. In order to do this one needs to be able to read the contents of a directory. Reading a directory is a bit like reading a file. One starts with opening the directory, reads from it, and then closes the directory. In the same folder as this document, you should find a directory called read_directory. We will use this directory as an example here.

We use the function called os.listdir(path) to open a directory where path is a string that shows the path to the desired directory. Before using the os.listdir() function one has to import a collection or library of functions called the os library. As an example:

```
import os    #import os library
path = "/home/read_directory"
dir_list=os.listdir(path)
```

dir_list is a list whose elements correspond to the name of a file in the directory specified by path.

Write a Python script that reads the folder called read_directory and prints out the list of files.

Answer:

```
#!/usr/bin/python
import os
path = "/home/read_directory"
dir_list=os.listdir(path)
for filename in dir_list:
      print "%s\n" %filename
```

The output is

```
fasta_seq1
fasta_seq2
readme
```

To get a list of files with a particular pattern in a directory, one can use the function glob.glob(”path with specified pattern”). (Don’t worry about the arcane names of some of the functions you encounter here.) To use glob, one has to import another library of functions called the glob library. In the above example, if one wants to list the files containing fasta in their filename, one can use the following program:

Answer:

```
#!/usr/bin/python
```

```
import glob
path = "/home/read_directory/*fasta*"
dir_list=glob.glob(path)
for filename in dir_list:
      print "%s\n" %filename
```

The output is

```
fasta_seq1
fasta_seq2
```

The * character in *fasta* stands for a wildcard character that can match any other character. Note that using *fasta* in the path serves to select only the files that contain fasta as a part of their name. If one chooses to select files that start with fasta or end with fasta, one should use fasta* and *fasta respectively.

16. A final exercise

You can now write a functional Python program that tests some of the key concepts you have learnt in the previous sections.

Exercise: Open the file titled fasta_sequence.txt in the same folder as the Python_trainer.doc. The FASTA format is a standard format for representing sequence information. A sequence in the FASTA format begins with a single line description, followed by lines of sequence data. The description line is distinguished from the sequence data by a greater-than (">") symbol at the beginning.

The file fasta_sequence.txt contains the amino acid sequence of two proteins, calmodulin and cytochrome b. Write a Python program that extracts the sequence of both proteins and saves them in two separate files in the FASTA format. The files should have the titles corresponding to the name of the proteins. This exercise will use some of the key concepts you have learnt in the previous sections, namely, regular expressions and extracting pattern.