Unix and Python tutorial with exercises for computational & systems biology

Sampsa Hautaniemi
sampsa@mit.edu
Modified by Sabrina L. Spencer and Laura Sontag

February 2, 2006

Contents

1 Hands on UNIX

Unix is a multi-user, multi-tasking operating system; hundreds of Unix users are able to use tens of program simultaneously. Accordingly, Unix provides computational biologists with many useful features. For example, high performance computing clusters, such as the New England Biogrid, are practically always based on Unix or Linux. Therefore, some familiarity with the Unix commands is indispensable for a systems biologist.

The Unix part of this tutorial is directed to students with no prior experience with Unix systems. The goal of this section is to review basic commands and principles for the Unix operating system, and using this tutorial a student should be able to:

- copy files in Unix.
- create directories and navigate in Unix.
- get help in Unix.
- execute programs, such as Python, in Unix.

More comprehensive texts are available, for example, in [Unix Tutorial, Fiamingo et al., 1998].

1.1 Basic UNIX operations

In this section, basic commands for Unix are reviewed. A command is a way to tell the Unix system to do a task. A command has the form

command [**options**] [**arguments**]

In general, a command may have several options. Options are usually preceded by a hyphen (-) and several options can be concatenated. Here, only the most useful options are discussed.

Unix is a rather old operating system (b. 1970). This brings forward several differences between the Unix systems and desktop/laptop computer operating systems, such as Windows and Apple. For example, commands in Unix are typed in command prompt (no need to grove for a mouse and click windows). Unix is also case-sensitive, so string "text" is different from "TEXT", and command "exit" is executed while "Exit" causes an error.

Throughout this tutorial, Unix commands are typed in bold italics (e.g. ***cd***), options in bold (e.g. ***ls*** **-la**), and examples of the Unix operations in Athena are typed with the `typewrite` font. Often, especially when using editors, some commands are preceded with pressing down control, alt or shift, these are denoted with ***CTRL***, ***ALT***, ***SHIFT***, respectively. For example, in order to exit a Pico editor session, one needs to first press down the control key and then "x", which is denoted as ***CTRL***+**x**.

1.2 Logging in and out UNIX

In order to use a Unix operating system one needs to log in. The easiest way is via a workstation, another option could be an SSH (telnet) program in a PC/MAC.

Whatever means are used to make the connection to Unix, you need to give your username and password in order to log in. If your username and password match an entry in the database of approved usernames and associated passwords, you have successfully logged in to the Athena cluster (which actually is a Linux system) and the command prompt appears:

```
athena%
```

In order to end a Unix session, the user needs to log out. This can done with the command ***exit*** (or with the slightly longer command ***logout***).

1.2.1 Help yourself

Unix contains a comprehensive reference manual on all commands and some programs in Unix. Access to this manual is obtained via command ***man***. For example, type ***man*** **exit** in order to get help for the command ***exit***.

At MIT, ***man*** is not the only source for rapid Unix help; Information Services and Technology (IST) offers consultation and maintains excellent pages on Athena [Athena Homepage]. In addition to the IST webpages, the command ***olc*** in Unix opens an on-line dialog with the Athena consults.

1.2.2 Directory and listing commands

The Unix file system is a tree-like structure, where the root is presented with a slash (/). After logging in Athena, you are located in your home directory. Command ***pwd*** shows current location in the directory tree. For example,

```
athena% pwd
/afs/athena.mit.edu/user/s/a/sampsa
```

Command ***mkdir*** (make directory) creates a new directory. For example, in order to create directory *BE490*, type ***mkdir*** **BE490** in the command prompt. command ***rmdir*** (remove directory) removes a directory, which must be empty (i.e., it should not contain any files or directories). For example, in order to delete the folder BE490, type ***rmdir*** **BE490**.

Command ***cd*** (change directory) enables navigation in the directory structure. For example, in order to move to directory *BE490*, type ***cd*** **BE490**. The directory must exist before executing ***cd***. In order to move one step towards the root, two dots (..) should be used. For example, if you are in the directory *BE490*, type ***cd ..*** in order to get back to your home directory. Typing ***cd*** ~ always brings you back to your home directory, not matter where you are in the directory structure.

Command ***ls*** (list) lists all files and directories in the current directory. ***ls*** has many options, but ***ls* -la** is particularly useful because it lists each file and directory in one row and shows their properties such as attributes, size and creation date. We will discuss attributes also in Section 1.2.4. Sometimes a directory contains so many files that they do not fit to the screen. In this case ***ls* -la** | **more** is useful.

1.2.3 File commands

Unix contains several ways to move and manipulate files. The most useful one in this course is ***cp*** (copy), which make a copy of a file to a specified directory. For example, assume we want to copy file *copyme.py* from directory
/afs/athena.mit.edu/user/s/a/sampsa/www/BE490/python
to
/afs/athena.mit.edu/user/s/a/sampsa/BE490

One way to do thus is as follows (␣ denotes a space):

```
athena% cp␣/afs/athena.mit.edu/user/s/a/sampsa/www/BE490/python/copyme.py
␣/afs/athena.mit.edu/user/s/a/sampsa/BE490/copyme.py
```

A much shorter way is to remember that ∼ denotes your home directory and in case you want the copy to have the same name, the name does not need to be repeated. Thus, the above command can be executed also:

```
athena% cp ∼/www/BE490/python/copyme.py ∼/BE490/
```

Files can be removed with the command ***rm*** (remove). For example, typing ***rm* copyme.py** removes *copyme.py.*

1.2.4 Security

Since Unix is a multiuser system, basic knowledge on some security issues is needed. Students using Unix need to be especially careful what they leave to their directories and how these directories are secured, in general. Unix gives many ways to deal with the security, such as ***chmod*** (permissions mode of a file). Attributes of a file or a directory can be listed with ***ls* -la**. For example,

```
athena% ls -la copyme.py
-rwxrwxrwx 1 sampsa mit 5 Jan 17 09:40 copyme.py
```

Entries starting from the second position (`rwxr`...) denote attributes for *copyme.py.* The first entry indicates if the listed entity is a file (`-`) or a directory (`d`). Entries 2–4 denote what rights the user has; `r` is read, `w` write, and `x` execute. Entries 5–7 denote `rwx` rights to a group (which needs to be specified) and entries 8–10 to all users. Thus, *copyme.py* can be read and altered by all the users.

Security issues above do not, however, directly concern Athena users. The reason is that Athena has the AFS file system (for more information, see [Working on Athena]). However, in different Unix systems not having the AFS (e.g. the New England Biogrid), it may be useful to check from time to time if the folders and files have appropriate security.

1.3 Executing programs in UNIX

Unix systems have several useful programs installed, such as Python, g++, MATLAB, LaTeX, etc. For example, if we want to execute *copyme.py* with Python (.py ending indicates that the file is a Python program), we simply write ***python*** **copyme.py** and Python interpreter executes the file:

```
athena% python copyme.py
This print is all I do.
Unless somebody has altered me (everybody is allowed to modify copyme.py).
athena%
```

In this case *copyme.py* was successfully executed and it printed two lines to the screen.

Another example of executing programs in Unix is sending email with program Pine. For example, ***pine*** **-attach copyme.py sampsa@mit.edu**, opens Pine program, *copyme.py* is attached and the recipient of the email is sampsa@mit.edu.

1.4 Editors

Unix offers a great variety of editors for typing and editing text. Probably the user-friendliest is Pico. Another widely used editor is Emacs, which is very flexible and valuable for heavy users. However, the flexibility of Emacs makes it far more difficult to use than Pico. For example, if you start Emacs and want to quit the program, that is done by typing first ***CTRL***+**x** and then ***CTRL***+**c**. In this course, no support for Emacs is provided but in addition to the Emacs tutorial (type ***CTRL***+**h t** in Emacs) there are many tutorials available, for example [Emacs Tutorial].

If your Unix experience consist only of a few hours, it is highly recommended that you choose Pico for this course. The main reason for this is that Pico is easy to use. If you want to edit *copyme.py*, just type ***pico*** **copyme.py**. If the file does not exist in the directory, Pico will create new file. A short Pico online tutorial is available at [Pico Tutorial].

1.5 Unix exercises

1. Make directory *BE490* in your home directory. (Section 1.2.2)
2. Move to directory `/afs/athena.mit.edu/user/s/a/sampsa/www/BE490/python/` and find out how many files there are. What attributes do these files have? (Section 1.2.2)

3. Copy file `/afs/athena.mit.edu/user/s/a/sampsa/www/BE490/python/Unix_intro.py` to your folder *BE490*. (Section 1.2.3)
4. Execute *Unix_intro.py* with Python in your directory *BE490*. (Section 1.3)
5. Open *Unix_intro.py* with Pico (or with an editor of your choice) and follow instructions in the file. Remember to save your changes. (Section 1.4)
6. Execute *Unix_intro.py* again with Python.
7. Email *Unix_intro.py* to yourself. (Section 1.3)

2 Introduction to programming and Python

Python is an *object-oriented*, *interpreted*, *programming language* mainly used for making *scripts* (terms in italic will be explained in this tutorial). The aim of this tutorial is to provide the students with sufficient knowledge of programming in Python to be able to solve basic bioinformatic tasks. Several excellent Python tutorials exist for beginners and advanced users, and it is highly recommended to use them in addition to this one [A Python Tutorial, Many Python Tutorials, Python book].

A programming language is a set of vocabulary and grammatical rules for instructing a computer to perform specific tasks. Python is a "high-level language", which means that its syntax resembles that of natural language, such as English. Even though Python is much more simple than English, it is much more versatile and easier to learn than low-level languages such as Assembler or actual machine language, which is understood by the computer. Accordingly, a high-level program must be translated to machine language. In general, the translation can be done in two ways. The first is to compile the program, which is how C++ works. The second is to interpret the program, which is how Python works. Compilation translates the whole program directly into machine language, while the interpreter translates one line at a time. Interpreted programs, and Python in particular, have several benefits, of which the most important is that the programs can be studied interactively.

A script (macro) is a sequence of commands that are interpreted. Scripting allows the users to make programs that automatically perform a specific task. For example, assume that you obtain hundreds of sequences and you want to execute BLAST to them. A simple Python script allows you automatically perform the BLAST and store the results, while your contribution is restricted to coding the script and starting it.

Python is an object-oriented programming language, which means that each data type can have several types of operations or methods. For example, a string object has several types of methods, such as 'count' and this function is used in Python (to open Python in Athena, type ***python***) as follows:

```
>>> "GATTACA".count("A")
3
```

We will explore this example in greater details in the subsequent sections. Object-oriented programming enables programmers to create modules, which is of great help in larger programming projects.

All functions and files mentioned in this tutorial are available at
`/afs/athena.mit.edu/user/s/a/sampsa/www/BE490/python .`
You can also download them at http://www.mit.edu/∼sampsa/BE490/python/
Feel free to use them as you like in this course.

2.1 Before starting programming

Programming is merely giving the computer instructions what to do. Therefore, it is imperative that the programmer knows what he/she is going to do before writing a single line of a program. That is, make sure that you understand the assignment before starting to program.

A fruitful approach to tasks requiring programming is top-down approach: First, understand what the programs needs to do; what are the inputs and what the program needs to output. Second, split the task into smaller tasks, such as "Read a file", "Find start/stop codon", "Read amino acids between the start and the stop codon", "Write the results". These smaller tasks correspond to modules (or functions) and they can be used in several programs doing similar things. For example, once you have made a function that reads a FASTA file, it can be used in many programs using FASTA files and with small changes to read other file formats.

2.2 Starting and ending Python session

It is recommended that you write the programs with an editor, such as Pico. The files can be executed with the command ***python*** **filename**. For example, the Python script for the first assignment for Prof. Yaffe would be *yaffe_1.py*. The file can be executed with the command ***python*** **yaffe_1.py**. As Python is an interpreted language, the command ***python*** **-i yaffe_1.py** opens an interactive mode, which enables debugging of the code. In order to quit interactive mode, press **CTRL+d**. Throughout this tutorial, interactive Python mode is denoted with `>>>` .

In this course, it is required that the programs can be executed with the command ***python*** **filename**. If you do your assignments in any other environment, such as in Windows, make sure that the program runs also in Athena.

2.3 Using lists, strings and dictionaries

Python has three basic data types; dictionaries, lists and tuples. Dictionaries are perhaps the most efficient and briefly introduced in Section 2.3.3. Due to educational reasons only lists in the context of strings are reviewed in detail. A string (a set of consecutive characters) can be defined with double (" ") or single quotes (' '). Throughout this tutorial we use double quotes:

```
>>> s = "GATTACA"
>>> s
'GATTACA'
```

Two or more strings can be concatenated easily using `+` sign:

```
>>> s = "GATTA"
>>> s2 = s + "CA"
>>> s2
'GATTACA'
>>> s3 = s + " " + "CA"
>>> s3
'GATTA CA'
```

Several strings, which do not need to be of same length, can be stored to lists. A list is delimited with brackets (`[]`). After creating a list, the entries can be retrieved easily using a bracketed index. However, perhaps somewhat counterintuitively, the first index to a list is 0, not 1. For example, in the interactive Python mode:

```
>>> li = ["GATTACA", "ATTAC", "ACATTAG"]
>>> li
['GATTACA', 'ATTAC', 'ACATTAG']
>>> li[1]
'ATTAC'
>>> li[0]
'GATTACA'
```

2.3.1 Basic list and string methods

As everything is an object in Python, a list has several methods that can be used to retrieve information and manipulate the stored strings. Command ***dir*** lists all methods for an object. For example, type ***dir(list)*** to get all methods for lists. Methods are called using period (`.`) and parenthesis (`()`). Another useful help command is ***help***. For example, ***help(list.extend)*** prints help for the list method 'extend'. Method 'extend' adds a list to the existing list:

```
>>> li = ["GATTACA", "ATTAC", "ACATTAG"]
>>> li.extend(["GAGA"])
>>> li
['GATTACA', 'ATTAC', 'ACATTAG', 'GAGA']
```

If you forget to type brackets, each entry is taken to be its own list:

```
>>> li.extend("GGG")
>>> li
['GATTACA', 'ATTAC', 'ACATTAG', 'GAGA', 'G', 'G', 'G']
```

The method 'find' finds the first entry of a substring. For example, starting position of the substring "TTA" for the first sequence in the list *li*:

```
>>> li = ["GATTACA", "ATTAC", "ACATTAG"]
>>> li[0].find("TTA")
2
```

If the desired string is not found, 'find' returns -1:

```
>>> li = ["GATTACA", "ATTAC", "ACATTAG"]
>>> li[1].find("GAT")
-1
```

Yet another example is 'count', which counts the number of a substring:

```
>>> li = ["GATTACA", "ATTAC", "ACATTAG"]
>>> li[0].count("G")
1
```

Methods for lists are sufficient for basic operations such as finding and counting. However, sometimes there is a need to more sophisticated string operations. Python allows users to import modules that can perform complex tasks. In the case of strings the next step after string and list methods is to import the module *string*. The string module can be imported with the command ***import*** and the operations are first preceded by the module from which they are imported:

```
>>> import string
>>> s = "GATTACA"
>>> string.count(s, "A")
3
```

For the full list of the operations in the string module, type ***dir(string)***, ***help(string)***, and/or visit Python online tutorial [String module commands].

In some cases the string module is not powerful enough. The next (and in some sense the ultimate) step is to use regular expressions (***import re***). The drawback of the regular expressions is that they are much more difficult to use than the string and list methods. Indeed, there are several tutorials for just how to use the regular expressions, e.g. [Regular expression Tutorial]. In this course you should be able to perform all programming tasks without regular expressions.

2.3.2 Indexing

Getting slices of a string is somewhat counterintuitive because the first index is included in the resulting slice, but the last one is not. Examples follow:

```
>>> s = "GATTACA"
>>> s[1:3]
'AT'
>>> s[0:2]
'GA'
>>> s[0:-1]
'GATTAC'
>>> s[1:len(s)]
'ATTACA'
```

Note that when using method 'len' the last entry is also included in the slice.

2.3.3 Dictionaries

Dictionaries are mapping constructs consisting of *key-value* pairs. A dictionary is created with curly brackets ({}) or the command ***dict***. For example, let's make dictionary that has two keys (*Seq1* and *Seq2*) and two values (GATTACA and ATTAC), then add the sequence ACATTAGA (key is *Seq3*) and observe that lists are not good keys:

```
>>> my_dict = {"Seq1":"GATTACA","Seq2":"ATTAC"}
>>> my_dict
{'Seq2': 'ATTAC', 'Seq1': 'GATTACA'}
>>> my_dict["Seq3"] = "ACATTAG"
>>> my_dict
{'Seq3': 'ACATTAG', 'Seq2': 'ATTAC', 'Seq1': 'GATTACA'}
>>> ListIsNotGoodKey = ["Seq4"]
>>> my_dict[ListIsNotGoodKey] = "ThisWontWork"
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: list objects are unhashable
>>> my_dict["Seq4"] = ["ButThisWill"]
>>> my_dict
{'Seq3': 'ACATTAG', 'Seq2': 'ATTAC', 'Seq1': 'GATTACA', 'Seq4':
['ButThisWill']}
```

Note that lists cannot be keys but can be values (the same applies also to dictionaries: dictionaries cannot be keys). Keys and associated values are initialized using a colon (:) and new entries are added just by using the key as above. Note also that the order of the key-value is irrelevant in a dictionary because the values are retrieved using the keys. For example, let's retrieve GATTACA:

```
>>> my_dict = {"Seq1":"GATTACA","Seq2":"ATTAC"}
>>> my_dict["Seq1"]
'GATTACA'
```

Dictionaries, like lists and strings have several methods. Use ***help(dict)*** to see the listing. For example, in order to store all the keys and values, the methods 'keys' and 'values' are useful. For example,

```
>>> my_dict = {"Seq1":"GATTACA","Seq2":"ATTAC"}
>>> keys = my_dict.keys()
>>> vals = my_dict.values()
>>> keys
['Seq2', 'Seq1']
>>> vals
['ATTAC', 'GATTACA']
```

2.3.4 How to make a copy of a data type?

Making copies of data types is slightly more complicated in Python than in the other programming languages. In Python there are two copy operations; *shallow* and *deep*. A shallow copy is actually a reference to the original data type, which means that altering elements in the copy alters the elements also in the original. If you use equal operator (=), the copy is a shallow copy. In order to make a deep copy, the module copy is needed (***import*** **copy**). A deep copy is independent from the original data type, meaning that altering elements in a deep copy does not alter the elements in the original. Some examples for lists follow:

```
>>> import copy
>>> original = ["GATTACA", "ATTAC", "ACATTAG"]
>>> shcopy = original # or shcopy = copy.copy(original)
>>> shcopy
['GATTACA', 'ATTAC', 'ACATTAG']
>>> del shcopy[1]
>>> shcopy
['GATTACA', 'ACATTAG']
>>> original
['GATTACA', 'ACATTAG']
>>> dpcopy = copy.deepcopy(original)
>>> dpcopy
['GATTACA', 'ACATTAG']
>>> del dpcopy[0]
>>> dpcopy
['ACATTAG']
>>> original
['GATTACA', 'ACATTAG']
```

2.4 Controlling the flow

In real-life programming tasks one wants to control the flow of the program. For example, IF a sequence has a start codon at the position *i*, THEN print the characters UNTIL stop codon is reached. A way to control the flow is via conditional execution (***if-elseif-then***) construction.

Controlling the flow requires the concept of *code indenting.* **Code indenting is very important in Python**! In this tutorial we use three spaces (␣␣␣) to denote an indented block, but any other number is as good as long as it remains consistent throughout the file. Indenting means that everything in ***if***, ***elif*** or ***else*** blocks needs to be indented. A colon after ***if***, ***elif*** or ***else*** statement tells that the block begins and returning indention up one level tells Python that the block has ended. An ***if***-statement does not necessarily need to be followed by ***elif***- or ***then***-blocks.

For example, assume we want first to compare if a sequence contains more or equal number of "A"s than another sequence and then print this information to the screen. When programs become more complex, it is better to write the program into a file (using an editor such as Pico) and then execute it using command ***python***. Here, we have typed the following to *iftest.py* (mark ␣ denotes a space):

```
s1 = "GATTACA"
s2 = "TATAG"
if s1.count("A") == s2.count("A") == 0:                               1
␣␣␣print "Strings", s1, "and", s2, "do not have any As"
elif s1.count("A") > s2.count("A"):                                   2
␣␣␣print "String", s1, "has more As than",s2
elif s1.count("A") == s2.count("A"):                                  3
␣␣␣print "Strings", s1, "and", s2, "have equal numbers of As"
else:                                                                 4
␣␣␣print "String", s2, "has more As than", s1
print "This line will be printed always"                              5
```

First, in $\boxed{1}$ we examine if the strings have at least one "A". Note colon (`:`) and indention! Note also that an ***if***-statement may have several comparisons. If either of the strings has at least one "A", in $\boxed{2}$ we compare if the number of "A"s is greater in *s1* than in *s2*. In $\boxed{3}$ we test if the number of the "A"s is the equal. If *s1* does not have greater or equal number of "A"s, then the only possibility is that *s2* has more, which is printed in the else block $\boxed{4}$. Regardless what happens in the ***if-elif-then*** blocks, $\boxed{5}$ will be executed always because it is not indented. Now, *iftest.py* is executed:

```
athena% python iftest.py
String GATTACA has more As than TATAG
This line will be printed always
athena%
```

In Python, character `==` denotes equal comparison and using `=` in comparison causes an error. Character `>` denotes "strictly greater than". "Greater or equal than" is compared with `>=`, while `a != b` compares whether `a` is different from `b`. It is also worth remembering that if an ***if*** or ***elif*** condition realizes, the rest of the ***elif*** conditions are not tested (in different programming languages this may be different).

2.5 Loops

Repeated execution allows programmers to perform the same operation for several variables. Repeated execution can be done using for-loops. For example, assume we have a list containing sequences and we want to print them one by one. Again we code using an editor, make file *fortest.py* and execute the program using ***python***:

```
li = ["GATTACA", "ATTAC", "ACATTAG"]
for i in li:
␣␣␣print i

print "Again this will be printed"
```

Now, *fortest.py* is executed:

```
athena% python fortest.py
GATTACA
ATTAC
ACATTAG
Again this will be printed
athena%
```

Note that it does not matter if there is an empty line between the for-block and the ***print*** command. This is true in general in Python; one can add as many empty lines to the program as needed (sometimes they improve readability of the program).

Often with for-loops the method 'range' is useful. Parameters for 'range' are **start** (optional), **end** (obligatory), and **step** (optional) and 'range' returns a list containing an arithmetic progression of integers. For example,

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(0,5,1)
[0, 1, 2, 3, 4]
>>> i = range(0,7,3)
>>> i
[0, 3, 6]
```

The method 'range' can be used also with strings:

```
>>> s = "GATTACA"
>>> i = range(0,len(s),3)
>>> i
[0, 3, 6]
>>> s[i[1]:i[2]]
'TAC'
```

The method 'range' is especially useful in for-loops (in interactive Python):

```
>>> s = "GATTACA"
>>> for i in range(0,len(s),3):
...␣␣␣print "now i = ", i, "and slice:", s[i:i+3]
...
now i = 0 and slice:  GAT
now i = 3 and slice:  TAC
now i = 6 and slice:  A
>>>
```

Note how Python does not care that we tried to access non-existing elements and just returned the last letter.

In addition to for-loop, one can use while-loops for repeated execution. A while-loop does something until a condition is fulfilled. It is useful when reading something from a file (more discussion on dealing with the files is offered in the next section). For example, if *Seq.txt* consists of sequences GATTACA, ATTAC, ACATTAG (one in each row), we could read them line by line (the following code is stored to *whiletest.py*):

```
f = open("Seq.txt", "r")
seq = f.readline()
while seq != "" :
␣␣␣print seq.strip()
␣␣␣seq = f.readline()
f.close()
print "Once again this will be printed"
```

Now, *whiletest.py* is executed:

```
athena% python whiletest.py
GATTACA
ATTAC
ACATTAG
Once again this will be printed
athena%
```

Method 'strip()' reads the sequence and removes all leading and trailing characters defined in the parenthesis. If the parenthesis is empty, like in the example above, then whitespace characters (spaces, tabs, carriage returns and line feeds) are removed. If 'strip' was not used, there would be an empty line after each sequence.

2.6 File reading and writing

In order for a script to be practical for real-life tasks, it must be able to read data from a file and write information to a file. Here basic file input/output (I/O) tools are reviewed. In general, there are three steps in the file handling:

1. Open a file.
2. Do something with the file (read or write).
3. Close the file.

Let's begin with opening the file. If you are just reading something from a file, there is no need to allow writing to the file. For example, assume there is a file named *Seq.txt* and you need to read data from there. The file is opened (and closed) as follows:

```
f = open("Seq.txt", "r")
<...place some really cool code here...>
f.close()
```

Using the command ***open***, you can open files in a Python program. Attribute `"r"` makes the file open in read-only mode, which means that you cannot (accidentally or deliberately) write anything to *Seq.txt*. Further, assume that you need to write the contents of *Seq.txt* directly (of course, in real-life you would do something sophisticated to the data before storing it again) to the file *output.txt*. First, we need again to open both files (and then close them afterwards):

```
f = open("Seq.txt", "r")
towrite = open("output.txt", "w")
<...place some really cool code here...>
f.close()
towrite.close()
```

If *output.txt* does not exist, attribute `"w"` creates a file and names it as *output.txt*, and assigns a write mode to it. If *output.txt* existed, the contents of it are deleted before new data are stored into it, so be careful when using `"w"`.

Now, first we read all the lines from *Seq.txt* and then use a for-loop to print each line to one row in *output.txt*. The code below is typed to *iotest.py*:

```
readfile = open("Seq.txt", "r")
writefile = open("output.txt", "w")

contents = readfile.readlines()
```

1

```
for i in contents:
␣␣␣SeqToWrite = i.strip()                                        2
␣␣␣print >> writefile, SeqToWrite                                3
readfile.close()
writefile.close()
```

In $\boxed{1}$ we read every line in *output.txt* using 'readlines' method. If you want to read only one line, the method would be 'readline'. In $\boxed{2}$ we read an entry from contents using the method 'strip()', which removes all leading and trailing white spaces from the sequence. In $\boxed{3}$ we print a sequence directly to the file *output.txt* using ***print*** and `>>` , which directs the data flow to the file. We could have also used 'write' method directly as follows:

```
readfile = open("Seq.txt", "r")
writefile = open("output.txt", "w")

contents = readfile.readlines()
for i in contents:
␣␣␣writefile.write(i)
readfile.close()
writefile.close()
```

Note that here we did not delete newline characters from the sequence. If we did, all the sequences would have been written together to the first row.

2.6.1 Exception handling

When dealing with the files, it may happen that the file to be opened using `"r"` does not exist or is corrupted. If this is not detected when opening the file, strange error messages may result later. Error handling in Python is done via exceptions, i.e. ***try***, ***catch*** commands.

The basic structure of an try-catch block is:

```
try:
␣␣␣<Do something, e.g.  open a file.>
except:
␣␣␣<Define here what to if an error occurred.>
<...here life may go on...>
```

Exceptions allow programmers to catch some anticipated errors and overcome them *a priori* so that the program does not crash when the error happens. Exception handling is a rather subtle issue and has to do also with style (it is more elegant if an error is detected and the program exists gracefully than the interpreter announces the error and halts). In this course only the basic exception operations are needed.

2.7 Functions

Functions allow the programmer to store valuable procedures and use them frequently by calling them in the main program. In Python, a function block is denoted with the command ***def***. For example, a function that takes in a name of a file, tries to open it, reads its contents to a list and returns the list, would be as follows (*functiontest.py*):

```
def ReadFileContents(filename):
␣␣␣try:
␣␣␣␣␣␣file = open(filename,"r")
␣␣␣except:
␣␣␣␣␣␣#Do nothing but return -1
␣␣␣␣␣␣return -1
␣␣␣contents = file.readlines()
␣␣␣file.close()
␣␣␣return contents

# Main program
filename = "Seq.txt"
sequences = ReadFileContents(filename)
print sequences
```

If *ReadFileContents* is not able to read the file, it returns -1. Since the main program only prints the sequences, in case the file could not be read, -1 is printed (Test this, e.g. rename *Seq.txt* to *Seq2.txt*). In real-life programs exceptions can be used to terminate the program and inform the user on the error. Note that in the *except*-block we have a comment. In Python, programmers can (and should) type comments using the hash character (#). Everything after the hash character is considered as a comment and will not be executed. Also, note that a function needs to be declared (declaration begins with ***def*** and ends when indentation gets back one to main level) before using the function. Since the function above did not use 'strip' method, the last character of each sequence is newline character (check this).

If a function returns something to the main program, this can be done with the command ***return*** which is followed by the returned variables that are separated with commas. It is not necessary, however, for a function to return anything.

2.7.1 Scopes and namespaces

When using functions, it is important to understand where a variable's values can and cannot be accessed or modified. In Python the rules for how and when variables' values can be accessed are described in terms of *namespaces* and *scopes* [Python book]. These concepts are rather subtle and it is strongly recommend to use a Python book or tutorial and get more thorough understanding in these issues.

When a variable is called (e.g. ***print*** **x**), Python needs to know the value of the variable (x). There are three namespaces in which Python searches the values: local, global and built-in namespaces. The first is local namespace, in which all bindings created in a block (e.g. function) belong. Blocks, such as functions, have unique local namespaces and thus one function cannot access the namespace of another function. So, if a variable is called in a function, the first place Python searches is the local namespace and if the variable (e.g. x) is defined there, Python uses that value. If, however, the variable is not defined in the local namespace, Python expands its search to the global namespace.

The global namespace contains bindings to all variables including function names defined in the file or module. When you start an interactive Python session, the global namespace is `"__main__"` . If the variable is not defined in the local namespace, Python searches the global namespace. If the variable is defined in the global namespace, Python uses this value. For example, assume you use variable x in a function but forget to define x to be an input argument (and you have not defined the variable elsewhere in the function). Now, if in the main program you also have a variable called x, the function will execute since it finds the value for x in the global namespace (i.e., in the main program). Let's see this with *scopetest.py*:

```
def testfun( y ):
␣␣␣z = "GATTACA"
␣␣␣print "This is x’s value:", x
␣␣␣print "This is y’s value:", y
␣␣␣print "This is z’s value:", z

# Main program
x = "ATTAC"
z = "ACATTAG"
testfun("GAGA")
print z
print y # This will cause an error
```

Now, let's execute *scopetest.py*

```
athena% python scopetest.py
This is x’s value:  ATTAC
This is y’s value:  GAGA
This is z’s value:  GATTACA
ACATTAG
Traceback (most recent call last):
File "scopetest.py", line 20, in ?
print y
NameError:  name ’y’ is not defined
athena%
```

So, the command ***print* x** in the function did not cause errors since x is defined in the main program. However, trying to print y in the main program caused an error because y is not defined. In other words, the scope of x is global (it can be seen in the main program and in the functions) but y is local (it is accessible only in the function *testfun*). Note that in the function the value for z is GATTACA, while in the main program it is ACATTAG.

After the global namespace comes built-in namespace and it contains several Python functions (e.g. ***range***, ***int***). The built-in namespace is created every time when the interpreter starts. Normally programs do not modify the built-in namespace.

2.8 Biopython

Biopython is freely available project consisting of tools for computational molecular biology [Biopython Homepage]. Detailed discussion on how to use Biopython and what is included in it is not covered in this tutorial. However, you can install Biopython to Athena as follows:

```
athena% add seven
athena% biopython
< Let's test if this worked >
>>> import Bio.Seq
>>> new_seq = Bio.Seq.Seq("GATTACA")
```

2.9 Help!?

Interactive Python has interactive help (***help***). In order to use help, one needs to know the module or data type for which help is needed. For example, in order to find out the methods for strings, type ***help(str)***, on lists type ***help(list)***, on dictionaries type ***help(dict)***, and so forth. In order to get slightly more detailed help on a specific method in a module, one needs to type the module first and then method. For example, ***readline*** is a method for the object file, and in order to get help on the command ***readline***, type ***help(file.readline)***. When getting help on the modules, remember to import the module first. For example, first ***import string*** and then ***help(string)***. More thorough help is available at Python webpages [A Python Tutorial].

2.10 Summary

- Before starting to code, make sure you understand the assignment.
- Use Pico (or Emacs) to write the code.
- Python is case-sensitive; “Print” is different from “print”.
- Use functions as much as possible (so you can reuse them later. . .)
- Declare functions before you use them.

- Comment your code (# character).

- Do not forget colon (:) after defining an if/for/def/etc. block.

- Remember indentation; every if/for/def/etc. block needs an additional indentation level. You can choose the level (e.g. three spaces), the main point is to keep it constant in the program.

- Use a interactive help and Python tutorial (http://www.python.org/) or a book.

- Have fun when coding.

3 Python programming task

Write a program that reads abstracts from a file, finds the amount of “the” articles in the abstracts and prints the counts to the file *output.txt*, so that the number of “the” articles in the first abstract is in the first row, the number of “the” articles in the second abstract in the second row etc. An example of the input file is

```
> Modeling of signal-response cascades using decision tree analysis
MOTIVATION: Signal transduction cascades governing cell functional responses to stimulatory cues
play crucial roles in cell regulatory systems ...
...
> An algebra of dimerization and its implications for G-protein coupled receptor signaling.
Many species of receptors form dimers, but how can we use this information to make predictions
about signal transduction?  ...
...
```

The abstract file you should use to test your program is located at:
`/afs/athena.mit.edu/user/s/a/sampsa/www/BE490/python/abstractfile.txt`

After you have written and executed the program, open a web browser and go to PubMed. In PubMed, find both articles and manually compute the number of “the” articles. Does it match the number you got?

3.1 Hints

This task consists of three steps:

- Read text from a file.
- Find the number of “the” articles.
- Write counts to a file.

So, it is convenient to make three functions that do the above mentioned tasks, and call them in the main program.

1. line = ff.readline() reads a line to variable line until the end of the file is reached (line != ””) (how about using ***while***?).
2. Each abstract begins with `>` , so `if line.startswith(">")` could be useful (it tests if a string starts with a character specified in the parenthesis).
3. If you operate with lists, the method ’extend’ is useful.

References

[A Python Tutorial] http://www.python.org/doc/current/tut/tut.html

[Athena Homepage] http://web.mit.edu/consult/www/

[Biopython Homepage] http://www.biopython.org/

[Python book] H.M Deitel, P.J. Deitel, J.P. Liperi, B.A Wiedermann *Python How to Program.* Prentice Hall, Upper Saddle River, NJ, 2002.

[Emacs Tutorial] http://www.lib.uchicago.edu/keith/tcl-course/emacs-tutorial.html

[Fiamingo et al., 1998] Frank G. Fiamingo, Linda DeBula and Linda Condron. *Introduction to Unix.* University Technology Services, The Ohio State University, 1998. http://wks.uts.ohio-state.edu/unix_course/unix.html

[Many Python Tutorials] http://www.python.org/doc/Intros.html

[Pico Tutorial] http://www.indiana.edu/∼ucspubs/b103/

[Regular expression Tutorial] http://www.amk.ca/python/howto/regex/

[String module commands] http://docs.python.org/lib/node107.html

[Unix Tutorial] http://www.helpfixmypc.com/unix/unixtutorial.htm

[Working on Athena] http://web.mit.edu/olh/Working/Working.html