CSE 20
Beginning Programming in Python
Lab Assignment 3

The goal of this project is to learn how to create a Python script, which is a Python program that does not require you to type `python3` at the Unix command line. In other words, you just type the name of the program to run it. Further, the program file name will not need to end in `.py`. You'll also learn about Unix *environment variables* (`PATH` and `HOME` in particular) and how to change their values.

To begin, log on to the timeshare **unix.ucsc.edu** (see lab) and type `which python3` at the command line. The result should be

```
$ which python3
/usr/local/bin/python3
```

In Unix, `which` gives the location of a command in the file system. The above output indicates that the `python3` command is located in the directory `/usr/local/bin`. Think of the Unix file system as a tree in which each directory is a branch point. The "full path" to the `python3` command is

`/`	the base of the file system, called the `root` directory
`usr/`	a directory containing read only data, including programs
`local/`	a directory containing programs local to this site, **unix.ucsc.edu**
`bin/`	a directory containing binary executable programs

The command `man hier` prints a detailed description of the Linux file system hierarchy. Try using `which` on different Unix commands, like `ls`, `cd`, `pwd` and `which` itself. You'll see that almost all of them are located in `/usr/bin`.

We see that Unix commands are programs stored in different locations in the file system. To run a command, you type the name of the program. But how does Unix know where to find all of these commands? The short answer is that it doesn't. Much of the behavior of your terminal session depends on the values of *environment variables* that are initialized when you log on. The environment variable that tells the system where to look for commands is called `PATH`. Type `echo $PATH` to see the value of your `PATH` environment variable. (Note the `$` in this command is *not* the Unix command prompt, and you *do* type it. Think of the `$` in `$PATH` as saying "value of".) Try `printenv` to see a listing of all your environment variables. You'll see `PATH`, along with another variable called `HOME` (and many others). The `HOME` variable contains the full path name to your home directory.

My `PATH` variable is printed below. Yours should be similar, but not identical.

```
$ echo $PATH
/usr/lib64/qt-3.3/bin:/opt/rh/llvm-toolset-7/root/usr/bin:/opt/rh/llvm-
toolset-7/root/usr/sbin:/opt/rh/devtoolset-8/root/usr/bin:/usr/local
/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/usr/local/texlive/2019/bin
/x86_64-linux:/usr/local/codebench/bin:/afs/cats.ucsc.edu/users/f/ptantalo
/bin:.
```

This is a single string with no line breaks, wrapped in the output. Here's a prettier version of the string.

```
/usr/lib64/qt-3.3/bin:
/opt/rh/llvm-toolset-7/root/usr/bin:
/opt/rh/llvm-toolset-7/root/usr/sbin:
/opt/rh/devtoolset-8/root/usr/bin:
/usr/local/bin:
/usr/bin:
/usr/local/sbin:
/usr/sbin:
/usr/local/texlive/2019/bin/x86_64-linux:
/usr/local/codebench/bin:
/afs/cats.ucsc.edu/users/f/ptantalo/bin:
.
```

We see that the `PATH` variable is a colon ":" separated list of Unix directories, each given by their full path name. Whenever you type a command at the `$` prompt, the command interpreter does a linear search of the `PATH` variable, looking within each named directory. The search stops when the command is found, or when there are no more directories to look in. If the command is found, it is executed. If it is not found, you'll get the error message "`command not found`". For instance, type `blah` at the prompt.

```
$ blah
bash: blah: command not found
```

If there had been a file named `blah` in any of the above directories, the command interpreter would have considered it to be a program, and tried to run it. If `blah` had existed in multiple directories, the interpreter would have executed the first one it found in the linear search. If file `blah` exists but is not a program, some other error message would be printed.

Let's look closer at the `PATH` variable above. Observe that it contains `/usr/local/bin`, the directory containing `python3`. Without this directory in my `PATH`, I could not use the `python3` command to run Python programs, or to start the Python interpreter. I could still do these things, but not by typing `python3`. Instead I'd have to type the full path name of the command: `/usr/local/bin/python3`. Now focus on the last two directories in the string.

```
/afs/cats.ucsc.edu/users/f/ptantalo/bin:
.
```

The first is a subdirectory of my home directory called `/bin`. If I write a program that I wish to run on a regular basis, this is where I put it. The second, which you may have overlooked since it is just a dot ".", is Unix shorthand for my *current working directory*, i.e. where I am now. Having dot "`.`" in my `PATH` allows me to `cd` into a directory and run any program located there. Without this, I would have to type `./prog` to run a program called `prog` in my current directory. Some users, as a personal preference, choose to include dot "`.`" in their `PATH`, and others do not.

When you examined your `PATH`, it's likely that the last two lines above were not in it. Create a subdirectory of your home directory called `bin` by doing (from within your home directory)

```
$ mkdir bin
```

then type `ls` to verify that the new directory was created. To alter your `PATH` variable, use the `export` command as follows, being sure to type it exactly as shown.

```
$ export PATH="$PATH:$HOME/bin"
```

As we saw, the HOME environment variable stores the full path name to your home directory. So the above export command assigns PATH to be the old value of PATH, followed by a colon ":", followed by the full path name to your newly created bin directory. Verify this by typing echo $PATH. Note that your PATH variable is only changed temporarily. If you log out, then log back in to the timeshare, PATH reverts to its original value.

If you want to permanently alter PATH, edit the file .bash_profile in your home directory and add the above line. If you want to also add the dot, set PATH to "$PATH:$HOME/bin:." Be careful to make no other changes to .bash_profile. In fact, it is recommended that you first back up the existing file by doing

```
$ cp .bash_profile .bash_profile.bak
```

With this done, file .bash_profile.bak contains the old version. After .bash_profile is edited, when you log out and log in, the new PATH persists.

All you need now are some programs to put into your bin directory. Save the files Echo and Dog from the webpage examples (/Examples/lab3) to your computer, then transfer them to the timeshare, and into your bin directory (see lab2). From within bin, make both files executable.

```
$ chmod u+x Echo
$ chmod u+x Dog
```

(See the man pages for a description of the chmod command. The variation chmod 755 also works.) Now from any directory, type Dog. Observe that the program prints a picture of a dog in a style called *ascii art*, which consists entirely of characters that you can type at a keyboard. Google the term "ascii art" to see many more examples. The program Echo emulates the Unix echo command. Try

```
$ Echo one two three
one two three
```

The second line above is the output of the program, which just repeats everything you type on the command line after Echo.

Both programs are in Python. Notice that in both cases the file does not end in .py, and we did not need to type python3. We include the source code for both programs here for reference.

```
#!/usr/local/bin/python3
#-----------------------------------------------------------------------
#  Echo
#  Emulates Unix echo command by printing command line arguments
#-----------------------------------------------------------------------
import sys
for w in sys.argv[1:]:
   print(w, end=' ')
# end for
print()

# end program ------------------------------------------------------
```

```
#!/usr/local/bin/python3
#-------------------------------------------------------------------------
#  Dog
#  Draws a picture of a dog
#-------------------------------------------------------------------------
s="""

       /^-^\
      / o o \
     /   Y   \
     V \ v / V
       / - \
      /     |
(     /      |
 ===/___) ||

"""
print(s)

# end program ------------------------------------------------------------
```

Observe that both programs begin with the same line

```
#!/usr/local/bin/python3
```

The sequence `#!` is known in Unix culture as *shebang*. It tells the command interpreter that this file is a script, i.e. a text file containing a program. After shebang, is the full path name to `python3`. Taken together, the above line tells the command interpreter to feed all the remaining lines to `python3`. This is why you don't have to type `python3`. That command is embedded within the file itself.

Now let's look at the rest of the `Echo` program. We import `sys`, a Python library containing commands that allow a program to interact with the system in which it is running. The expression `sys.argv` is a list containing everything on the command line that started the program. Index 0 in this list contains the name of the program `Echo`, and indices 1 through $\text{len}(\text{sys.argv}) - 1$ contain everything else on the command line. One more bit of syntax you may not have seen. If L is a list, then $L[a:b]$ is called a *slice*. It is the sublist of L consisting of all elements from index a (inclusive) to b (exclusive). The expression $L[a:]$ is the sublist consisting of all elements from indices a to the end. Thus sys.argv[1:] is a list of all words on the command line except the name of the program. The loop in `Echo` prints out these words, separated by spaces. See also the program `Args.py` in `/Examples`.

The `Dog` program is much simpler. We define a string containing the ascii art, then just print the string. The string must contain some newline characters. To define a multiline string, we surround it by triple quotes. Everything from """ to """, including the newlines, is included in the string.

Use what you've learned to write a Python script called `Box` (not `Box.py`) that prints out a variable sized piece of ascii art. Several test runs on the timeshare are included below.

```
$ Box 3 5

 +------+
 |      |
 +------+
```

```
$ Box 5 3

 +--+
 |  |
 |  |
 |  |
 +--+

$ Box 5 5

 +------+
 |      |
 |      |
 |      |
 +------+

$
```

Observe the blank lines before and after the art, and also the column of spaces immediately to the left of the art. Program `Box` takes two command line arguments, giving the height and width of the art, respectively. The height is measured in the number of lines printed, but the width is not the number of columns. This is because the height and width of an ascii character are not equal. A call to Box $a\ b$ will print a rectangle whose top and bottom consist of a "+", followed by $2(b-2)$ dashes "-", followed by a "+". The interior $a-2$ rows will consist of a vertical bar "|", followed by $2(b-2)$ spaces " ", followed by a vertical bar "|". The whole rectangle is moved one space to the right, giving the required blank column on the left. These specifications are designed to make the rectangle approximate a square when $a=b$. This is illustrated in the following runs.

```
$ Box 2 2

 ++
 ++

$ Box 3 3

 +--+
 |  |
 +--+

$ Box 4 4

 +----+
 |    |
 |    |
 +----+

$
```

All of this can be accomplished by writing some carefully designed loops. Your program will only be tested on integer inputs in the range 2 to 40. Place `Box` in your `bin` directory. You must be able to run `Box` from any directory, not just the one in which it is placed. Therefore, you must set your `PATH` variable to include `$HOME/bin`, as explained above. Create a directory called `lab3` within your `cse20` directory (which you created in lab2). Transfer the program `TestBox` (posted on the webpage in `/Examples/lab3`) to your

`lab3` directory. Note that this program is a Bash shell script, not a Python program. Now `cd` into `lab3` and do

```
$ chmod 700 TestBox
```

then

```
$ TestBox
```

The effect of the last command is to create a new file in your `lab3` directory called `TestBoxOut`. The contents of `TestBoxOut` will verify that you followed all instructions correctly. Submit your program `Box` and the file `TestBoxOut` to Gradescope before the due date.

This is certainly the hardest lab assignment so far, and almost qualifies as a programming assignment. As usual, if you have any problems, get help from TAs, Tutors and myself.