CSC/CPE 101: Spring 2018

LAB-1

Working with Linux environment:

1. Login to Linux environment:
   login: <enter username>
   Password: <enter password>
2. Open a terminal window. To do so, from the system menu on the desktop toolbar, select **Applications → System Tools → Terminal**. The Terminal program will present a window with a command-line prompt. At this prompt you can type Linux commands to list files, move files, create directories, etc. For this lab you will use only a few commands. Additional commands can be found from a link on the course website.
3. In the terminal, type **ls** at the prompt and hit <Enter>. This command will list the files in the current directory. (In some environments, e.g., Windows, a directory is commonly referred to as a folder.)
4. If you type **pwd**, the current directory will be printed (it is often helpful to type **pwd** while you are navigating directories). If you type **tree**, then you will see a tree-like listing of the directory structure rooted at the current directory.
5. Create a new directory for your coursework by typing **mkdir CPE101**. Use **ls** again to see that the new directory has been created.
6. Change into this new directory with **cd** by typing **cd CPE101**. To move back "up" one directory, type **cd ..**.

To summarize:

```
cd ~
mkdir CPE101
chmod go-rx CPE101
cd CPE101
```

7. Create a directory as LAB1

```
mkdir LAB1
cd LAB1
```

Editor:

There are many options for editing a Python program. On the department machines, you will find vi, vim, emacs/xemacs, nano, gedit, and some others. The editor that one uses is often a matter of taste. You are not required to use a specific editor, but we will offer some advice (and we will try to help with whichever one you choose). An editor is a personal choice - ask your friends which one(s) they use!

If you decide to connect to the department machines from home, then you'll want an editor that can work via a remote connection without much effort (i.e., without installing and running additional software). This makes gedit a less than desirable candidate (unless you want to learn two editors). The others will work fine, though you'll see some differences between xemacs and emacs (which is what you'd use remotely).

vi and emacs are powerful editors once you have learned how to use them. But they require some initial effort to learn their command sequences. Knowledge of one of these will, however, serve you well for quite some time.

nano (or pico, when nano isn't available) is very simple. It will feel, in some respects, like using Notepad. It is quick to learn and easy to use. Some people will mock others for using nano (some people are elitist dorks). You might choose to start with nano and then switch to one of the other editors as you "outgrow" nano (the others provide features that can aid you when programming).

A note concerning tabs: Whichever editor you choose, you should lookup how to convert tabs to spaces so that your Python source files do not contain a mix of tabs and spaces. **DO NOT use tab characters in your files.**

In general, whichever editor you choose, you should invest some time early to learn how to navigate quickly within a file. More specifically, you will want to learn how to jump to a specific line, to search within the file, and copy/paste lines.

If you cannot decide, then use nano. You always have the option of switching at a later time.

8. You will start the editor by typing the name at the prompt followed by the name of the file that you wish to edit. For instance, **nano -Ec lab1.py** (note the **-Ec** is useful to 1) convert tabs to spaces (E) and 2) direct nano to display the line number for the cursor's current position(c)).
9. A Python program is written in a plain text file. The program can be run by using a Python interpreter.
10. Save this file as lab1.py
11. Note: the files are accessible through PolyLearn.
12. To execute the program, type **python lab1.py** at the command-line prompt.
13. Using your editor of choice, modify *lab1.py* to replace **World** with your name and fill out the header comment at the beginning of the file. Execute the program to see the effect of this change.
14. The Python interpreter can be used in an interactive mode. In this mode, you will be able to type a statement and immediately see the result of its execution. Interactive mode is very useful for experimenting with the language and for testing small pieces of code.
    a. Type python and press enter the >>> prompt will appear in your screen.
    $ python
    >>>
    b. Now try some arithmetic statements and see the results
    c. Type each of the following (hit enter after each one) to see the result. When you are finished, you can exit the interpreter by typing ctrl-D (i.e., hold the control key and hit d).
        - 0 + 1
        - 2 * 2
        - 19 // 3
        - 19 / 3
        - 19 / 3.0
        - 19.0 / 3.0
        - 4 * 2 + 27 // 3 + 4
        - 4 * (2 + 27) // 3 + 4

d. Type python3 and press enter the >>> prompt will appear in your screen.
   $ python3
   >>>
e. Try 19//3, 19/3, and 19/3.0 and check the difference with previous results.
f. What is different between / and // operator?

Many people tend to focus on writing code as the singular activity of a programmer, but testing is one of the most important tasks that one can perform while programming. Proper testing provides a degree of confidence in your solution. During testing you will likely discover and then fix bugs (i.e., debug). Writing high quality test cases can greatly simplify the tasks of both finding and fixing bugs and, as such, will actually save you time during development.

For this part of the lab you will practice writing some simple test cases to gain experience with the unittest framework. Using your editor of choice, open the *lab1_test_cases.py* file. This file defines, using code that we will treat as a boilerplate for now, a testing class with a single testing function.

In the *test_expressions* function you will see a single test case already provided. You must add additional test cases to verify that the following expressions (exactly as written) produce the values that you expect. Note that you will want to use assertAlmostEqual instead of assertEqual to check computations that are expected to result in a floating point value.

- 0 + 1
- 2 * 2
- 19 // 3
- 19 / 3
- 19 / 3.0
- 19.0 / 3.0
- 4 * 2 + 27 // 3 + 4
- 4 * (2 + 27) // 3 + 4

Remember: use **assertAlmostEqual** when comparing floats, use **assertEqual** when comparing integers.

1. Save the following program as **python file lab1_test_cases.py**
   Note: the files are accessible through PolyLearn.
   or you can copy and paste the code in your editor and save as **lab1_test_cases.py**

```
# Test cases example for lab 1.
import unittest      # import the module that supports writing unit tests
# Define a class that will be used for these test cases.
# This class will include testing functions.
# Much of this code should be viewed as boilerplate for now.
#
class TestsLab1(unittest.TestCase):
  def test_expressions(self):        # Defines one testing function.
    self.assertEqual(0 + 1, 1)
    # Add code here (like the line above) for the additional test cases.
    self.assertEqual(2 * 2, 4)
    self.assertAlmostEqual(19/3.0, 6.333333333333333)
```

```
# Run the unit tests.
if __name__ == '__main__':
  unittest.main()
```

Run the program by typing **python3 lab1_test_cases.py**. You should now see a report of any tests that did not succeed.

2. Raise your hand as soon as you finished the program
3. Submit your work in PolyLearn
4. If you have time do more practice on Linux commands:
    a. l shows list of files
    b. ls –directory listing
       ls [-l] [-s] [-a] [dir or file names…]
       -l long listing
       -s include file sizes
       -a all files
    c. Copy a file
       cp –copying files
       cp <original file> <new file>
       Example:
           $ cp file1 file2
    d. Rename a file
       mv –rename files
       mv <original filename> <new filename>
    e. Delete a file
       rm –remove files
       rm <option> <file>
               -f force
               -i interactive (ask before removing)
               -r recursive
    f. Search
       grep –searching files
       grep <string> <filename>
       Example: File homework1.txt contains:
               This is line1.
               This is line2 for CSC101.
               This is line3.
       grep cs390 homework1.txt
       Result:
               This is line2 for CSC101.

More Linux command can be found in Linux documents and online resources.