Lab 1, CSC 101

Welcome to CSC/CPE 101. This first lab will help you to familiarize yourself with the programming environment provided in the department labs. This is the environment in which you will be working all quarter long, so take this time to ask questions and to get comfortable with the lab setup.

Environment

The lab machines run a distribution of the Linux operating system. For simplicity, and to gain experience in a, potentially, new environment, we will do our coursework in this environment.

Unix

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.

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.) 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.

Create a new directory for your coursework by typing **mkdir cpe101**. Use **ls** again to see that the new directory has been created.

Change into this new directory with **cd** by typing **cd cpe101**. To move back "up" one directory, type **cd ..**.

To summarize

- **ls** list files in the current directory
- **cd** change to another directory
- **mkdir** create a new directory
- **pwd** print (the path of) the current directory

Though these basic commands are enough to continue with this lab assignment, you should consider working through a Unix tutorial (many can be found on the web) at a later time.

Executing a Program

Download the lab1.zip file. Place this file in the **cpe101** directory created above; this can be done via the browser (by selecting the location to save to), via the graphical file manager, or through the use of the **mv** command in the terminal window (e.g., from the **cpe101** directory, after downloading, type **mv ~/Downloads/lab1.zip .**). If you need assistance doing this, please ask.

Unpack this file by typing **unzip lab1.zip** at the command prompt. This will create a directory named **lab1**; change into this directory by typing **cd lab1**. If you now list the contents of the **lab1** directory, you should see the following files: lab1.py lab1_test_cases.py. In addition, you will find the following subdirectories: `line vehicle`.

A Python program is written in a plain text file. The program can be run by using a Python interpreter.

You can see the contents of *lab1.py* by typing **more lab1.py** at the prompt.

To execute the program, type **python lab1.py** at the command-line prompt.

To summarize

- **mv** move files
- **unzip** extract contents of a **.zip** file
- **more** display contents of a file
- **python** python interpreter used to execute a program by specifying name of program file

Editing

It is suggested that you use the PyCharm Integrated Development Environment (IDE). It provides enhanced control of files during development. IDEs are used in professional development and make many of the tasks involved in writing code much easier. You will be given support to start using PyCharm, but first try the PyCharm quick-start tutorial provided by Jet-Brains.

To Do: 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.

Interactive Interpreter

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, but your general development process with be editing and executing a file as discussed previously.

Start the interpreter in interactive mode by typing **python** at the command prompt. You should now see something like the following.

```
Python 2.6.6 (r266:84292, Jan 22 2014, 09:42:36)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The `>>>` is the interpreter's prompt. You can type an expression at the prompt to see what it evaluates to. 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

Test Cases

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 # this test case is already provided
- 2 * 2
- 19 // 3
- 19 / 3
- 19 / 3.0
- 19.0 / 3.0
- 4 * 2 + 27 // 3 + 4
- 4 * (2 + 27) // 3 + 4

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

Compound Data -- Classes and Objects

In this part of the lab you will write new code. You are asked to complete the definition of two classes.

The following steps will walk you through writing the class definition and test cases for the first class. You will then repeat the process for an additional class. Pay careful attention to each of these steps as you will repeat them throughout the quarter.

Line class

In the *line* subdirectory, perform each of the following steps.

Implementation

In the *line* subdirectory, create a file named *line.py*. This is referred to as a "source" file. In this file you will provide the definition of the line class.

Create a class named `Line`. Within this class you will define the `__init__` function (note that there are two underscore characters before `init` and two after) to take, in this order, `self`, `x1`, `y1`, `x2`, and `y2` as arguments. This function must initialize the corresponding fields in `self` with the values of `x1`, `y1`, `x2`, and `y2`.

Testing

In the provided *line_tests.py* file, edit the *test_line* function to create a *Line* (with values for the `x1`, `y1`, `x2`, and `y2` arguments of your choice) and then test that each field was properly initialized.

Write a second test case by adding a *test_line_again* function to create and verify a second

Line.

You will also need to add `import line` to the top of this file. Try it out. If you cannot get it working, ask for assistance.

Type **python line_tests.py** to run the test cases.

vehicle class

As you did for `Line`, provide a class definition and test cases for a `Vehicle` in the *vehicle* directory. A `Vehicle` value must keep track of the number of wheels, the amount of fuel remaining, the number of doors, and whether or not the vehicle has a roof. Give some thought to the types of values that you will use for each of these attributes.

In *vehicle_tests.py*, write the test cases for a `Vehicle` to verify that each field is properly initialized.

Handin

Labs should be submitted on PolyLearn by the due date for credit.

Tutorial

All done? Great. Consider using any remaining time by working through a Unix tutorial. This is a good time to get more familiar with the environment and with your editor.