Lab 1, CSC 101

Welcome to CSC/CPE 101. This first lab will help you to familiarize yourself with the programming 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

We will be using PyCharm (link provided on Canvas) as the main programming environment. It is an integrated development environment (IDE), which includes, an editor with syntax highlighting and code completion capabilities, an interpreter (Python execution environment), a debugger (a tool to help you find errors in your code) and more. Download PyCharm and install it on your machine. Make sure you also download Python (3.7 or later) and install it. PyCharm will find Python on your machine and will use it (or a copy of it) to run your programs.

If you prefer to remote login to the campus lab machines, you're welcome to do that. You'll need to be familiar with Unix. Below is a brief 'getting started' with the Unix 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 lab files from the provided link on Canvas. Place these files 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

There are many options for editing a Python program. If you're using PyCharm, just open the desired file from the file/open menu and start editing.

On the department machines, you will find vi, 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).

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 (or avoid tabs altogether).

In general, whichever editor you choose, you should invest sometime 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.

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)).

To Do: 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.

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 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 16:30:00) [MSC v.1900 64
bit (AMD64)] on win32
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.

Demonstration

Demonstrate *lab1_test_cases*, the *line/line_tests*, and the *vehicle/vehicle_tests* programs to your instructor to have this lab recorded as completed. In addition, be prepared to show your instructor the source code for your line and vehicle programs.