Lab 1: An Introduction to Linux and Python

CSE/IT 107

NMT Computer Science

1 Introduction

The purpose of this lab is to introduce you to the Linux environment (including some common, useful commands) and to the Python programming environment. In this lab, you will learn how to

1. Log into Linux at the Tech Computer Center (TCC).
2. Learn to use IDLE as an interpreter.
3. Edit, compile, and run a few short Python programs.
4. Create a tarball and submit a lab to Moodle.

Throughout this semester's labs, we will be using specially formatted text to aid in your understanding. For example,

```
text in a console font within a gray box
```

is text that either appears in the terminal or is to be typed into the terminal verbatim. Even the case is important, so be sure to keep it the same! If a $ is used that indicates the terminal prompt. You do not type the $.

```
Text that appears in console font within a framed box is
sample Python code or program output.
```

Text in a `simple console font`, without a frame or background, indicates the name of a file, a function, or some action you need to perform, but not necessarily type into the terminal. Don't worry: as you become familiar with both the terminal and Python, it will become obvious which is being described!

2 Instructions

The following instructions will guide you through this lab. Because this is an introductory lab, the instructions are rather detailed and specific. However, it is important that you understand the concepts behind the actions you take—you will be repeating them throughout the semester. If you have any questions, ask the instructor, teaching assistant, or lab assistant.

2.1 Log in

To log into a TCC machine, you must use your TCC username and password. If you do not have a TCC account, you must visit the TCC Office to have one activated.

1. First, sit down at a computer. If you're looking at some flavor of Linux (such as Fedora), skip forward to Step 6. If you're looking at Windows, press `Ctrl+Alt+Delete`.
2. Click on the `Shutdown...` button.
3. Select `Restart` from the menu and click `OK`. Wait for Windows to shutdown and for the computer to restart.
4. When the computer reboots, you will be presented with a menu to select the operating system you prefer. Select `Fedora` or the flavor of Linux installed on the computer you are using.
5. Wait while the Linux operating system boots.
6. When the login screen comes up, type your username and your password followed by `Enter`.

It should be noted that most of these instructions will work on Windows or Mac OS with Python installed, though we recommend using Linux simply to gain familiarity, as later Computer Science classes will require it.

2.2 Open IDLE

Throughout this semester, we will primarily be working with IDLE. IDLE is an integrated development environment included with every Python installation. In this lab we will cover how to use it as a calculator, as well as how to use it to write simple programs.

3 Python as a calculator

Forgot ever using your calculator! Python rocks as a calculator.

When you first open IDLE, you should see something like this

```
Python 2.7.6 (default, Feb 26 2014, 12:07:17)
[GCC 4.8.2 20140206 (prerelease)] on linux2
Type "copyright", "credits" or "license()" for more information.
>>>
```

At the Python prompt `>>>` you can type in Python statements. For instance, to add 2 + 3

```
>>> 2 + 3
5
>>>
```

Notice how after a statement is executed, you are given a prompt to enter another statement. The `+` is the addition operator. Other basic math operators in python include:
`-` Subtraction: subtracts right hand operand from left hand operand

```
>>> 2 - 3
-1
```

`*` Multiplication: Multiplies values on either side of the operator

```
>>> 2 * 3
6
```

`/` Division, `//` Floor division - Divides left hand operand by right hand operand

Python 2

```
>>> 2 / 3
0
>>> 2 // 3
0
>>> 2 / 3.0
0.6666666666666666
>>> 2 // 3.0
0.0
>>> 2. / 3.
0.6666666666666666
>>> 2 / float(3)
0.6666666666666666
```

Python 3

```
>>> 2 / 3
0.6666666666666666
>>> 2 // 3
0
>>> 2 / 3.0
0.6666666666666666
>>> 2 // 3.0
0.0
>>> 2. / 3.
0.6666666666666666
>>> 2 / float(3)
0.6666666666666666
```

Notice the difference between Python 2 and 3 when using `/`. When Python 2 divides two integers using `/`, the decimal portion of the answer is dropped. If you want the answer you expect from a typical calculator when you divide, you have to change one or both of the numbers to a floating point number. There are a number of ways to have a floating point number in python. Add a decimal point or use the built-in class `float(x)`, where x is a number or a string. The zero after the decimal point is not required. Some programmers add it as a matter of style as it clearly identifies the number has a decimal point and is a floating point number.

In Python 3, using `/` will always include the decimal portion of the answer. If you wish to drop the decimal portion, you must use `//`. Dropping the decimal portion of the answer is called integer or floor division, since your answer will always be an integer. In this class, we will always use `/` to refer to floating point division and `//` to refer to integer division. If you are using Python 2, be you will need to be aware of the inconsistent behavior of `/`.

% Modulus: Divides left hand operand by right hand operand and returns the remainder.

```
>>> 5 % 3
2
>>> 3 % 5
3
```

** Exponent: Performs exponential (power) calculation on operator. `a ** b` means a raised to b.

```
>>> 10 ** 5
100000
>>> 5 ** 10
9765625
>>> 9 ** .5
3.0
>>> 5.5 ** 6.8
108259.56770464421
```

Besides fractional powers, python can handle very large numbers. Try raising 10 to the 100th power (a googol)

```
>>> 10 ** 100
10000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000L
```

The `L` at the end of the number indicates it is a long. Long integers are integers that are to big to represent with an integer data type.

Of course these commands can be combined to form more complex expressions. They follow typical precedence rules, with `**` having higher precedence than `*` `/` `%` `//`, which have higher precedence than `+` `-`. If operators have equal precedence they associate left to right. You can always use parenthesis to change the precedence.

```
>>> 6 * 5 % 4 - 6 ** 7 + 80 / 3 # ((6 * 5) % 4) - (6 ** 7) + (80 / 3)
-279908
>>> 6 * 5 % 4
2
>>> 6 * (5 % 4)
6
>>> (6 * 5) % 4
2
```

The `#` indicates a comment. Comments are ignored by the interpreter.

Now the real power of python as a calculator comes when you add variables and make assignments.

```
>>> x = 7
>>> y = 3 * x ** 2 + 7 * x + 6
>>> y
```

```
202
```

4 Running Python programs

So far, you have been running Python interactively. This is a nice feature as it lets you test ideas interactively. However, there are time you want to save your ideas to a text file and to run the code as a program.

Open a text editor (e.g. gedit) and create a one line python program:

```
print 'hello, world!\n'
```

The \n prints a newline. Save your file as `hello.py`. The `py` extension indicates it is a Python file.

Now in IPython, you can use the command `run` to execute the program. If you don't add a path before the file name, the file is assumed to be in your current working directory (cwd). If the file is in a different directory just navigate to it or use a path.

```
In [112]: run hello.py # assumes hello.py in cwd
hello, world!

In [113]:
```

From the terminal, you can do the same thing:

```
$ python hello.py
hello, world!
```

Make sure you can run `hello.py` from IPython and the terminal.

At times, you would like to run the Python file as a regular *nix program. To do this, you have to change the permission of the file so it is an executable.

```
$ chmod 755 hello.py
```

Where `chmod` changes the permission of the file. The numbers come from a combination of `r = 4, w = 2` and `x = 1`. To turn on `rwx` you add the numbers $4 + 2 + 1 = 7$. If you only want the file to be read only you would use 4. If you want it to be read and write you use 6. For read and executable you use 5. The first number sets the user's permission (7 in this case) the next number sets the group's permission (5 in this case), and the last number sets other's permissions (5 in this case). If you run the command `ls -l` the file's permission should be `rwxr-xr-x`.

Now one would think, you should be able to execute the file. Try it.

```
$ ./hello.py
```

You should get an error. The error is that the python program does not know where to find the python interpreter. You have to add the following line to `hello.py`. It also must be the `first` line in your program.

```
#!/usr/bin/env python

print 'hello, world!\n'
```

Now the program will run as a *nix executable. Try it.

Adding structure to your programs

While you can just string together python statements into a source file and run it, this strategy is soon to lead to ”spaghetti” code: a tangled mess of unmaintainable code. To prevent this we are going to use the following boilerplate code to write programs.

```
#!/usr/bin/env python

def main():
    print 'hello, world!'


# Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

Writing code in this way then starts the program with the `main()` function as other programming languages do. We will find later that it also allows you to import your code easily and makes it easy to test your code. You should create a file called `boilerplate.py` and then when you want to create a new python file, just copy `boilerplate.py` to a new file. For instance, if you want to create a new python file called `dna.py`, you would run the command:

```
$ cp boilerplate.py dna.py
```

5 ASCII Art

Copy `boilerplate.py` to `ascii.py`. The goal of the program is to create ASCII art that produces the word “python” across the terminal.

```
 1234567890123456789012345689012345678901234567890123456789
1                             #
2                             #
3                             #
4                       #     #
5                       #     #
6 ###     #       #  #######  ####      ####   ######
7 #  #     #     #      #     #   #    #    #  #    #
8 #   #     #   #       #     #    #   #    #  #    #
9 #  #       # #        #     #   #    #    #  #    #
0 ###         #         #     #  #      ####   #    #
1 #          #
2 #         #
3 #        #
4 #       #
5 #      #
```

The line and column numbering is not part of the program. They are there as a guide. Feel free to design your own typography, within the following constraints: letters are at most 10 characters wide; at most 10 lines high; and there is no more than 15 lines total; the terminal is 80 characters wide.

Python has some nice string features. Strings can be either enclosed in single or double quotes. For this program, use single quotes for your strings. To print 10 hash tags (#) rather than writing

```
print '##########'
```

you can “multiply” strings:

```
print 10 * '#'
```

To concatenate strings you use the + operator. To print 10 spaces, followed by 10 hash tags, followed by 20 spaces, followed by 5 hash tags, you do the following:

```
print 10 *  ' ' + 10 * '#'  + 20 * ' ' + 5 * '#'
```

6 The Tunnel

Copy `boilerplate.py` to a file named `tunnel.py`.

Download the files `tunnel` and `tunnel2` from Moodle. To terminate the programs enter CTRL + c. Run the programs:

```
$ ./tunnel
$ ./tunnel2
```

You should see a tunnel being created across the screen (`tunnel`). The other program produces two tunnels across the screen `tunnel2`.

Your job is to create these two programs as Python programs. The terminal is assumed to be 80 characters wide. The tunnel is 5 spaces wide and where it starts varies from line to line, but there is always a path through the tunnel. The print statement is similar to what you just did, printing out python. However, you need to add some randomness to the line you are printing. To get you started, experiment with the following command on the python interpreter.

```
In [171]: import random

In [172]: random.randrange(0, 10)
Out[172]: 2

In [173]: random.randrange(0, 10)
Out[173]: 8

In [174]: random.randrange(0, 10)
Out[174]: 6
```

To write an infinite loop you can write `while 1:`. To terminate the program use CTRL + d. If you want to slow the program down you can add these lines to your program:

```
import time
time.sleep(0.5)
```

If you want to speed up the program, comment out the line `time.sleep` or change the number to be smaller. Larger numbers will make the program slower.

Note: import statements occur at the beginning of the source code files. The following code prints out random integers between 0 and 9 until the user enters CTRL + d.

```
#!/usr/bin/env python

import random
import time



def main():

    while 1:
        x = random.randrange(0, 10)
        print x
        time.sleep(0.5)
```

```
#Standard boilerplate to call the main() function to begin
# the program.
if __name__ == '__main__':
    main()
```

7 Creating Tar archives

Tar is used much the same way that Zip is used in Windows: it combines many files and/or directories into a single file. Gzip is used in Linux to compress a single file, so the combination of Tar and Gzip do what Zip does. However, Tar deals with Gzip for you, so you will only need to learn and understand one command for zipping and extracting.

In the terminal (ensure you are in your `lab1` directory), type the following command, replacing `firstname` and `lastname` with your first and last names:

```
tar czvf cse107_firstname_lastname_lab1.tar.gz *.py
```

This creates the file `cse107_firstname_lastname_lab1.tar.gz` in the directory. The resulting archive, which includes every python file in your `lab1` directory, is called a tarball.

To check the contents of your tarball, run the following command:

```
tar tf cse107_firstname_lastname_lab1.tar.gz *.py
```

You should see a list of your Python source code files.

Upload your tarball in Moodle