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 not executed 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
```

Now enter `Ctrl + d`. This will exit the python interpreter.

4 IPython

The standard python interpreter has some limitations. From now on you are going to use IPython. A more sophisticated python interpreter.

From the command line type the following:

```
$ ipython --pylab
Python 2.7.3 |Anaconda 1.4.0 (64-bit)|
(default, Feb 25 2013, 18:46:31)
Type "copyright", "credits" or "license"
for more information.

IPython 0.13.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??'
for extra details.

Welcome to pylab, a matplotlib-based Python environment
[backend: Qt4Agg].
For more information, type 'help(pylab)'.

In [1]: x = 9

In [2]: x
Out[2]: 9

In [3]:
```

IPython replaces the python prompt `>>>` with `In [1]:` where the number represents the line number.

IPython provides a much richer environment to work in. Among other things, it understands *nix commands. At the IPython prompt, type in the commands `pwd,` `ls`, and `cd` to your home directory and back to your previous working directory using `cd -`. Note: your line numbering for the IPython prompt will differ.

```
In [26]: pwd
Out[26]: u'/u/jdoe/cse107/lab1'

In [27]: ls

In [28]: ls -a
```

```
./  ../:w

In [29]: cd
/u/jdoe

In [30]: pwd
Out[30]: u'/u/jdoe'

In [31]: cd -
/u/jdoe/cse107/lab1

In [32]: pwd
Out[32]: u'/u/jdoe/cse107/lab1'

In [33]: ls -alF
total 8
drwxrwx--- 2 jdoe 9876 4096 Sep  2 16:46 ./
drwxrwx--- 3 jdoe 9876 4096 Sep  2 16:46 ../

In [34]:
```

As the initial screen suggested. To get help with IPython type ?

```
In [35]: ?
```

You will get a screen of information. The cursor is at the bottom of the screen. It is flashing after the colon. Typing `h` will bring up a help screen on how to navigate the pager. Note: you can type `h` with any pager program like `less` or `man` from the command line. You will discover the commands are consistent across paging programs.

When you are done reading the IPython Introduction. Quit and return to the IPython prompt. Type `quickref` at the prompt. The same pager commands work as before.

IPython lets you access Python's built in help. Type `help()` at the prompt. This changes the prompt to `help>` to indicate you are using help, which is Python's built in documentation system. Follow the prompts. You should be able to get help on 'modules', 'keywords', or 'topics'.

```
In [7]: help()

Welcome to Python 2.7!  This is the online help utility.

If this is your first time using Python, you should definitely
check out the tutorial on the Internet at
http://docs.python.org/2.7/tutorial/.

Enter the name of any module, keyword, or topic to get help
on writing Python programs and using Python modules.  To quit
```

```
this help utility and return to the interpreter, just type "quit".

To get a list of available modules, keywords, or topics,
type "modules", "keywords", or "topics".  Each module also
comes with a one-line summary of what it does; to list the
modules whose summaries contain a given word such as "spam",
type "modules spam".

help> keywords

Here is a list of the Python keywords.
Enter any keyword to get more help.

and                 elif                if                  print
as                  else                import              raise
assert              except              in                  return
break               exec                is                  try
class               finally             lambda              while
continue            for                 not                 with
def                 from                or                  yield
del                 global              pass

help> and
```

After typing `and` you are given information about the keyword `and`. You navigate the page like before.

Another feature of IPython, is you can learn about any object using ? or ??. For example,

```
In [9]: x = 3

In [10]: x?
Type:       int
String Form:3
Docstring:
int(x=0) -> int or long
int(x, base=10) -> int or long

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is floating point, the conversion truncates towards zero
If x is outside the integer range, the function returns a long instead.

If x is not a number or if base is given, then x must be a string or
Unicode object representing an integer literal in the given base.
The
literal can be preceded by '+' or '-' and be surrounded by whitespace.
```

```
The base defaults to 10.  Valid bases are 0 and 2-36.  Base 0 means to
interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4

In [11]:
```

To get more information about an object you can use `??`. Sometimes there is not more information and you get the same output as `?`.

If you just want the *type* of the Python object, use the `type` builtin. You often need to know the type of the variable.

```
In [19]: type(x)
Out[19]: int
```

And if you want information on type use the command `type?` or `help(type)`.

A few other things about IPython. It supports the use of the up and down arrow keys to walk you through your command history. CTRL + a to move to the beginning of a line. CTRL + e to move to the end of a line. IPython also supports tab completion.

At the prompt, type `math.` followed by a TAB. Note the period after the word `math`. This is an example of tab completion.

```
In [19]: math. <TAB>
math.acos        math.degrees     math.fsum        math.pi
math.acosh       math.e           math.gamma       math.pow
math.asin        math.erf         math.hypot       math.radians
math.asinh       math.erfc        math.isinf       math.sin
math.atan        math.exp         math.isnan       math.sinh
math.atan2       math.expm1       math.ldexp       math.sqrt
math.atanh       math.fabs        math.lgamma      math.tan
math.ceil        math.factorial   math.log         math.tanh
math.copysign    math.floor       math.log10       math.trunc
math.cos         math.fmod        math.log1p
math.cosh        math.frexp       math.modf

In [19]: math.
```

You are presented with a list of availbale math functions. Use help to determine what these math functions do. For example, `help(math.cos)`

The math functions are built in. To use them you do this:

```
In [64]: x = 100

In [65]: log10(x)
Out[65]: 2.0
```

Of course these can be combined to make more complicated expressions:

```
In [66]: sqrt(log10(x))
Out[66]: 1.4142135623730951
```

One of the advantages of using IPython is has builtin support for matplotlib, which allows you to plot math functions.

Lets say you want to plot the line $y = x$ with $x \in [0, 10]$. In IPython, you would do the following:

```
In [71]: x = range(11) # why is it 11?

In [72]: help(range)

In [73]: y = x

In [74]: plot(x, y)
```

This should launch a new window with a plot of $y = x$. Try to plot the line $y = x$ for x from -100 to 100. Read the help page about `range` to figure out how. If you do not close the plot window or use the command `clf()` or `close()` your plot will just be written to the current plot window.

Let's try to plot something more complicated, $y = x^2$. Lets plot this over the values of x $\in [-10, 10]$

```
In [77]: x = range(-10, 11) # Why is it 11 and not 10?

In [78]: x
Out[78]: [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1,
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] #note the [], a list

In [79]: y = x ** 2
---------------------------------------------------------------------------
TypeError
Traceback (most recent call last)
<ipython-input-79-f4db1f0b84fe> in <module>()
----> 1 y = x ** 2

TypeError: unsupported operand type(s)
for ** or pow(): 'list' and 'int'

In [80]:
```

The problem is you are mixing types: a *list* with an *int*. A list in Python is a "comma-separated values (items) between square brackets". The method `range()` returns a list of numbers. In this case a list of integers from -10 to 10. The line `y = x ** 2` assumes x is of type `int`, which it is not and that is why you get a `TypeError`. The `Traceback` gives the stack frame that created the error.

To fix this, we need to make `y` a list as well.

```
y = [i ** 2 for i in x]
```

The `[]` indicates that the object you are creating is a list. Inside the brackets we put an expression, `i ** 2`, followed by a for loop. Python's for statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. To see what the for loop does, type the for loop at the IPython prompt. When you type a for loop you have to finish the loop body before the code is executed. That is what the ....: indicates. When you are done typing the loop body, hit Enter and the for loop is executed.

```
In [96]: for i in x:
   ....:     print i,
   ....:
-10 -9 -8 -7 -6 -5 -4 -3 -2 -1 0 1 2 3 4 5 6 7 8 9 10
```

That is a comma after the `i`. Without the comma, python will print each value with a new line. Try it.

As you can see `for i in x` justs walks through the list, printing out the value of `i`. In the assignment for the variable `y` we just add an expression `i ** 2` that is applied to each `i`.

Try this at the prompt:

```
In [100]: for i in x:
    print i ** 2,
   .....:
100 81 64 49 36 25 16 9 4 1 0 1 4 9 16 25 36 49 64 81 100
```

Now you just need to assign each value to the variable `y`. At the prompt type this:

```
In [101]: y = [] #create a empty list

In [102]: for i in x:
   .....:     y.append(i ** 2)
   .....:

In [103]: y
Out[103]: [100, 81, 64, 49, 36, 25, 16, 9, 4, 1, 0, 1, 4, 9,
16, 25, 36, 49, 64, 81, 100]

In [104]: type(y)
Out[104]: list
```

This code creates an empty list and then adds each element to the of the list (`y.append()`) with the square of each x value. Run the command `help(y.append)`. The statement `y = [i ** 2 for i in x]` is just shorthand for the above for loop. Pretty cool!

To correctly plot $y = x^2$ then, you would do the following:

```
In [108]: close() #close the figure window

In [109]: x = range(-10, 11)
In [110]: y = [i ** 2 for i in x]

In [111]: plot(x, y)
Out[111]: [<matplotlib.lines.Line2D at 0x45828d0>]
```

Try to plot the following functions:

$y = x^3$ for $x \in [-10, -10]$

$y = 4x^2 + 5x + 6$ for $x \in [-10, -10]$

$y = sin(x)$ for $x \in [-10, 10]$.

The sine plot doesn't plot very well as it uses integer data. To fix this, there is a function `frange()` which is similar to `range()`. The `frange()` requires that you provide a start, stop, and a step (delta). For instance, setting `x = frange(-10, 10, .1)` creates an numpy array (more on numpy later in class) of floating point values from -10 to 10 separated by a tenth. Plot sine using these x values. A much smoother curve is produced. If you do not know what sine is, ask the Instructor.

Do something similar to plot the function:

$y = sqrt(x)$ for $x \in [0, 100]$

Come up with a few functions on your own to plot.

5 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
```

6 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 * '#'
```

7 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()
```

8 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