CSci 1133
Lab Exercise 1

Getting Started

In this first lab exercise, you will explore a number of computational resources that will be needed during the course of the upcoming semester. You will also begin writing your first Python code!

Logging-in

To log in from any of the CSE computer labs (such as KHKH 1-262) pick an available Linux machine and type your UMN *username* and *password* in the login box. If this is not successful, you may need to manually register your account. To register your account, have someone open a web browser for you and go to:

https://wwws.cs.umn.edu/account-management

The rest is pretty self-explanatory. If you are not a College of Science and Engineering student, you will need to provide the name and userid of a "sponsor". Ask one of your TAs for this information. Note that it may take 30 minutes or more for your account to be opened so, in the meantime, you should complete this lab with another student who has an active CSE-Labs account.

Register your GitHub account

We will be using the GitHub software repository manager throughout the semester for submitting programming assignments and other details. Don't worry about the particulars just yet, it will all be explained in upcoming labs.

For now, you simply need to log into the online GitHub manager to identify yourself. Using a web browser, navigate to:

https://github.umn.edu

The GitHub manager will ask you for your username and password. Use your UMN x500 ID (Internet ID) and password. After you have confirmed that you have access, you can close the web browser.

Part 1: Brief UNIX Introduction (adapted from C. Swanson, "Unix Tutorial")

The UNIX operating system dates to the early 1970's and remains one of the most efficient, robust and reliable software systems in use today. The "operating system" refers to the collection of programs residing on your computer that provide the interface to the hardware of the machine. The operating system is responsible for maintaining and organizing data on external devices, loading and running application programs, etc. You are probably familiar with graphically based operating systems such as Microsoft Windows, Apple Mac OS and others (actually, Mac OS uses the UNIX operating system for its "core" functions). Our lab computers are running a derivative of the original UNIX operating system known as Linux; specifically the Ubuntu distribution of Linux and, as computer scientists, you will need to understand how to use the UNIX "command line" interface. The following brief introduction will guide you through a few essentials. You are encouraged to explore and learn more on your own! There is an abundant wealth of UNIX/Linux information and tutorials available on the Internet.

A Brief History of the UNIX Terminal
In the early days of UNIX, communication with the operating system was performed via an electro-mechanical contraption known as a "teletypewriter terminal" or "terminal" for short. It had a keyboard for generating "input" to the operating system and a paper printer for producing "output" from the operating system.

The concept and function of the "terminal" still applies today, but now we *emulate* the teletype device using a graphical display window. When we run the "terminal" application, our keystrokes are input directly to the operating system and the subsequent output is displayed in the application window rather than being printed on a roll of paper.

Why use the UNIX Terminal?
The terminal is still in wide use today because of its power and efficiency. With experience, a wide range of actions can be performed quickly in one place without wasting time clicking through windows. *Scripts*, which are stored sets of commands, can be created to automate repetitive tasks e.g., renaming a large directory of files, checking the status of a large network of computers, or converting and combining video files.

To run programs and generally interact with the operating system in "command line" mode, you must first start the terminal application by clicking on the terminal icon (generally located on the upper left side of your display -- it looks like a little computer monitor). Start the terminal now (seek help from your TAs if you need it).

The Command Line Interface
Unlike many other computer systems you may have used in the past, you will interact with UNIX mainly using typed commands. The commands are typed into a terminal window along with any required or optional *flags* or *arguments*. *Arguments* are additional options and information that tell the computer what to do with the command. For example, if you want to rename a file, you need to provide the name of the file to be renamed. This way of communicating with UNIX is referred to as the *Command Line Interface*, or CLI.

After launching the terminal, the operating system will prompt you for a command, using a short text string like this:

```
username@machinename:~$
```

where *username* is your UNIX user name and *machine* is the name of the computer you are currently logged in to. The '$' symbol (it may also be some other symbol such as '%') lets you know that the system is ready to read a command. The text in between the `:` symbol and the `$` symbol tells you where you are on your computer (your *current working directory*).

Basic Commands
You tell the operating system what you would like it to do by typing a short *command* and pressing the ENTER key. Most UNIX commands are shortened names that describe what they do. For example, `cp` stands for *copy* and `mv` stands for *move*. Commands can be edited while they are still being typed using the left and right arrow keys or the backspace key. UNIX commands must be typed *precisely*, and are case-sensitive.

Type the following command after the prompt (note: [ENTER] represents the "ENTER" key on the keyboard):

```
ls [ENTER]
```

The `ls` command is short for "list" and will list all the files in the directory (folder) you are currently "in". To see what directory you are in (*print working directory*) type:

```
pwd [ENTER]
```

Unless you have already been playing around (good for you!), you should have seen:

```
/home/username/
```

where `username` is the user name you logged in with. This is your "home directory" which we will explain in a bit.

For a listing of more UNIX commands see: https://en.wikipedia.org/wiki/List_of_Unix_commands

Getting Help

If you ever don't know or forget what a particular command does or how it is used, you can use the UNIX `man` command to learn more. `man` stands for *manual* page and will tell you everything you need to know (and more) about a particular UNIX command. Simply type: `man` *command-name* [ENTER] to display the manual page, and press [SPACE] to scroll through the pages of information. To see the man page for the `ls` command type:

```
man ls [ENTER]
```

The command name you typed after `man` was an *argument* to the `man` command. A very useful *flag* for the `man` command is the `-k` flag, which allows you to search for manuals on commands containing some search text. To search for commands dealing with Python type:

```
man -k python [ENTER]
```

This will dump a large list of commands (and more!) that deal with Python. Reading the `man` page on `man` is a good place to start if you want to understand more about this list!

```
man man [ENTER]
```

Files and Directories

If you've used a modern computer system, you are probably familiar with the concepts of files and folders. A file is a collection of data that has a specific name associated with it. A folder is a special file that contains zero or more files or other folders. In UNIX, folders are called *directories*, but they are essentially the same thing. Files and directories allow users to store and organize their data in an easy-to-use fashion.

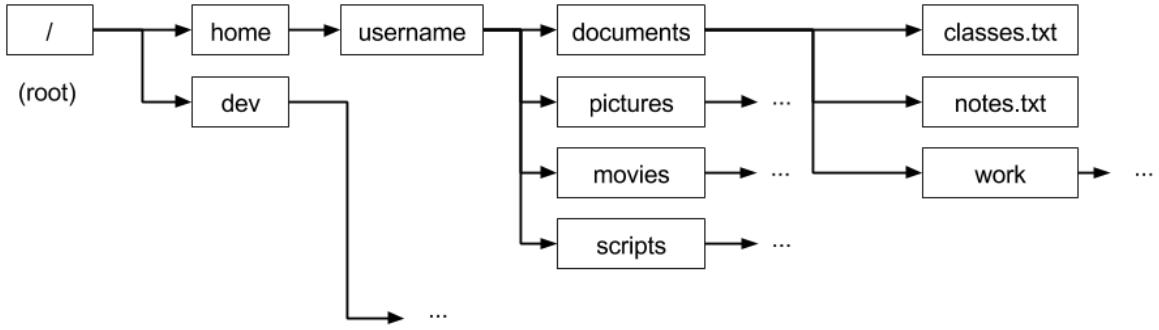
File and directory names are case-sensitive, and special care should be taken when naming them. Avoid using spaces or special characters except for the underscore, dash, and period. Certain special characters are forbidden in file names, but others are allowed and can cause problems. Also, avoid giving files the same name as UNIX commands.

All of your personal files and directories are contained within your *home* directory. When you first start a new terminal, your *home* directory will always be your (active) *working* directory. The working directory is simply the directory you are currently viewing, or working in, since you can only be "in" one directory at a time.

You can use the `pwd` (print working directory) command at any time to show you what your current working directory is. This will print the complete *path* to your current working directory. The path is "where" the file or directory is located relative to the root directory. For example, the path to a file named `classes.txt` might look like this :

```
/home/username/documents/classes.txt
```

This *path* describes the location of the specific `classes.txt` file in the context of the whole file system, for example:



Each section of the path is a directory and directories are separated by forward slash characters. The `ls` command from earlier lists all of the files and directories in your working directory, so if your current directory was "documents" in the above structure, entering the `ls` command would display:

```
classes.txt
notes.txt
work/
```

NOTE: For the next few commands, you may want to open the graphical file manager for an analogy of what is happening in an environment you may be more familiar with. (The file manager is generally located on the upper left side of your display -- it should look like a little file folder or a filing cabinet). If the file manager does not open to your home directory automatically, navigate to it. Do you notice any similarities between what is displayed in the graphical interface and what the `ls` command prints out?

Modifying and Navigating the File System

Directories can be created and destroyed in the file system using `mkdir` (*make directory*) and `rmdir` (*remove directory*). Try making a new directory in your home directory:

`mkdir hello` [ENTER]

You can change to a different working directory with the *change directory* command, `cd`. To change to your newly created `hello` directory:

`cd hello` [ENTER]

Now we will introduce some special names for directories. The double-dot (`..`) stands for the *parent* directory, which is the directory that *contains* the current working directory. We can use this to "move up" one level. Try

```
cd .. [ENTER]
pwd   [ENTER]
```

Also, the tilde (~) is shorthand for your *home* directory. We could have also returned to the home directory (from anywhere) by typing:

```
cd ~ [ENTER]
```

Lastly, the single-dot (`.`) stands for the current directory. All of these names can be used in parts of larger paths. For example, the path:

```
/home/username/hello
```

is the same as:

```
~/hello
```

Now that we have returned to the home directory, we can remove our hello directory. Type

```
rmdir hello [ENTER]
ls [ENTER]
```

Note that `rmdir` will only remove empty directories (that is, directories that contain no files or other directories). To remove directories containing files, the `rm` command is needed. Check out its `man` page.

Creating Terminal Scripts

As mentioned earlier, scripts can be created using the command line which allow you to combine and save sets of instructions. For example, the `ls` command does not say what the current working directory is by default, but `pwd` does. We could combine the two commands in a script and then we would only need to run the script to see both!

To create the command line script we need to create a new text file. There are many ways to create text files in UNIX, but for writing small program scripts it is convenient to use a text editor. A text editor is a relatively simple program that works like a stripped-down word processor. It allows you to enter/modify text and save the text to a file. There are a number of text editors available on CSELabs machines. We recommend that you use the Atom text editor because of its ease of use, and its advanced features that will come in handy later on when programming.

To open Atom, enter the following command in the terminal:

```
atom . [ENTER]
```

The dot (`.`) tells Atom that you want to open it in the current directory. The first time Atom is run, it will display several "Welcome" tabs. You may close all of these tabs, and press `Ctrl-n` (or [File → New File] in Atom's menu bar) to edit a blank file. It may be helpful to place the terminal and Atom side-by-side on your screen so that you can edit files and run them without having to change windows (click and drag the title bar all the way over to the left or right side of the screen). Once Atom is running, enter the following lines *exactly as shown* into the editor:

```
echo "--- Working Dir ---"
pwd
echo "--- Contents ---"
ls
```

Simply typing text in the editor program does not actually create the file; you must explicitly *save* the text to the file. To save your changes in Atom, type `Ctrl-s` on the keyboard, or press [File → Save] in Atom's menu bar. Atom will bring up a dialog box asking where to save your file and what to call it. Save it as `show.sh`, in the current directory. Don't close Atom yet, you'll need to use it later in the lab.

You have now created your first terminal script! To run a script you have created from the command line, you type `./scriptname`, where scriptname is `show.sh` in this case.

You will most likely get an error if you try to run the script right away. This is because we need to tell the UNIX operating system that this is an "executable" file, or a program that it can run. To do so, we use the command `chmod` with the `+x` flag and the file name as the argument:

```
chmod +x show.sh
```

Now try running the script with:

```
./show.sh
```

You should have seen the output from both the `pwd` and `ls` commands displayed together. Now, let's create a new directory for the scripts we make in this lab in the home directory. Enter the following commands, one after another:

```
cd ~ [ENTER]
mkdir myscripts [ENTER]
cd myscripts [ENTER]
```

We'll use the *copy* command, `cp`, to copy files from one directory to another. The copy command takes the form

```
cp source-file destination-file
```

where *source-file* is the name of the file to be copied, and *destination-file* is where we want to put it. Use `ls`, `pwd`, and `cd` to figure out where you saved your show.sh script, and then copy it to the new folder we created for it:

```
cp show.sh ~/myscripts/ [ENTER]
```

Now use the *list* command to list all of the files in the `myscripts` directory and make sure our script was copied successfully:

```
ls ~/myscripts/ [ENTER]
```

Once you have verified that you copied the file correctly, you can delete the file in the `myscripts` directory using the *remove* command. Be careful though! Unlike other computer systems, there is no "undelete" or "Recycle Bin" in UNIX. Type:

```
rm show.sh [ENTER]
```

Once you delete a file, it's gone for good in most cases. Now return to your home directory:

```
cd ~[ENTER]
```

While copying `myscripts` is an okay directory name, we will be having many labs, so it might be a good idea to have separate folder for each lab. We can rename directories and files using the *move* command:

```
mv myscripts hello [ENTER]
ls [ENTER]
```

Change it back using:

```
mv hello myscripts [ENTER]
ls [ENTER]
```

Now rename the folder containing your script to `lab1`.

Lastly, we will discuss two more convenient functions provided by the command line. These are *command history* and *tab completion*.

Command History
You may have noticed already that pushing the up arrow in a terminal window brings up previous commands. You can use this to your advantage to recall any command you have typed in that terminal window. This is especially useful if you make a mistake in a long command. You can simply press the up arrow until you arrive at that command, and correct the error using the left and right arrow keys. Try it:

```
makdir hello [ENTER]
```

Press the up arrow once to recall the command. Then press the left arrow until the cursor is on the ‘k’. Then press backspace to remove the erroneous ‘a’. Finally, press [ENTER] to retry the command. The cursor does not need to be at the end of the command; the terminal will take the whole line as the command regardless of cursor position.

Tab Completion
Often, you will have long filenames that are annoying to type. Tab completion will save you some typing. You can simply type the first few letters of a file or directory name, press the Tab key, and the terminal will try to complete the name for you. Try this:

```
cd ~ [ENTER]
cd h [TAB]
```

The terminal should "fill in" the rest of hello. But what happens when multiple files start with the same characters? Try this:

```
cd ~ [ENTER]
mkdir hydrogen [ENTER]
cd h [TAB]
```

The terminal should display the names of all files and directories that start with h. In this case, hello and hydrogen both qualify. Type

```
y [TAB]
```

and the terminal should fill in the rest of hydrogen.

You've learned to use several important UNIX commands for manipulating files. If you are still unsure of what you are doing, discuss it with one of your Lab TAs before continuing.

Part 2: Running Python

The best way to learn Python is to just start playing with it. But we should first take a minute or two to better understand what is going on behind the scenes...

A modern electronic digital computer is an automated *machine*, much like a player-piano, that 'executes' a series of instructions intended to accomplish some computational task. Individual instructions are executed in a sequential fashion, one instruction at a time. At the machine (hardware) level the instructions are taken from a set of instructions that is proprietary to a specific computer. Each instruction in the set represents a single *operation* that the machine will dutifully execute before proceeding with the next one. Programs written using these "low-level" instructions are referred to as *machine code*. *Machine code* is tedious to produce, prone to human error and, when completed, generally not portable to different computers. For these reasons, the vast majority of programs are written in *high-level* languages such as Python. Programs written in high-level languages are easier to write, comprehend and debug, and they are much more portable than native *machine code*. However, there is a price to be paid for this flexibility: the high-level language must be first translated into low-level machine instructions in order to be executed by a specific machine.

This translation can be accomplished *statically* (in advance of program execution) or *dynamically* (during program execution). Each of these methods had advantages and disadvantages, but both require a specific translator program to accomplish the task. Static language translators are referred to as *compilers*, whereas dynamic translators are generally called *interpreters*. Python uses a dynamic translation approach that requires us to first start the *interpreter* program in order to run a program written in the Python language.

In UNIX, we run programs in the same way as scripts or simple commands: simply type the name of the program and press [ENTER]. For example, to start the Python *interpreter* program we type:

```
python3 [ENTER]
```

NOTE CAREFULLY: There are two versions of the Python language and, unfortunately, they are not compatible with one another. ***We'll be using Python version 3 throughout our course, so you will need to be especially careful that you are running the correct version.***

Go ahead and try it. Be sure to type `python3` and not `python` or you will get the wrong version! You should see something like:

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more
 information.
>>>
```

The new prompt `>>>` indicates that you are now providing Python commands to the interpreter and not to the Unix command-line interface. The "interpreter" program that is now running is quite simple in concept: you simply type a Python expression and press [ENTER]. Python "interprets" the expression you entered, *evaluates* it and prints out the result of that evaluation.

The simplest expression we can provide is simply a number. Try this:

```
>>> 42 [ENTER]
42
```

Python dutifully *evaluates* the expression (determines its value) and reports that the value of "42" is 42 (duh!). That's great, but not very impressive... we can construct more complex Python expressions using mathematical operations in much the same way we write arithmetic expressions in mathematics. In general, an "expression" is a single input statement consisting of a series of "operands" and "operators":

```
>>> 42+20 [ENTER]
62
```

in this case, the "operator" is the familiar "plus sign". An operator *represents* some sort of atomic computer operation. We are familiar with the operation represented by the "plus" character in this case: it's binary addition, a + b. The *operands* are the values 42 and 20, so the interpreter causes the computer to execute an addition operation on the values 42 and 20 producing the 'result': 62.

We can combine any number of operands and operators into a single expression. Type something like:

```
>>> 4+5*2 [ENTER]
```

and Python will respond with the value of the expression:

```
14
```

Pretty cool! Now try some arithmetic expressions on your own. Note that +, - and / all have the usual meanings (addition, subtraction, division), but multiplication is represented with the * symbol.

We can save the result of an expression evaluation for use later on by simply giving it a name and using the assignment operator (=). Try this:

```
>>> x = 23.5 * 46 [ENTER]
```

Note that the "value" of the result was not shown… it was instead stored in the computer's memory and given the name: 'x'. If we want to know the value of x, we can ask the interpreter to tell us:

```
>>> x [ENTER]
1081.0
```

Note a subtle, but *very important*, distinction here. The ' = ' operation represents *assignment*, not the familiar "x is equal to 23.5 times 46". It means "*assign* the label x to the result of 23.5 times 46". For example, try this:

```
>>> x = x + 1 [ENTER]
>>> x [ENTER]
1082.0
```

Mathematically, it is nonsense to suggest that "x is equal to x + 1". But this makes perfect sense in Python, because it is not stating mathematical equivalence. The expression means: "associate the name x with the new value obtained when you add 1 to the current value of x" and is a mechanism that we will use frequently.

In general, to obtain the value of any named *variable* for use in another expression, we simply write its name in place of the value:

```
>>> x**2*3.14159 [ENTER]
3677934.81116
```

Note that the two asterisks (**) is not a mistake. This is how you represent exponentiation in Python. You may find a more frequent use for the value 3.14159. So you can just name it as well:

```
>>> pi = 3.14159 [ENTER]
```

Another way to see what the value of the object named 'pi' is, is to use a built-in Python *function* to print its value to the terminal display. Type the following:

```
>>> print(pi) [ENTER]
```

Python will print out the value of the object named pi, 3.14159.

Go ahead and explore on your own! Try creating more complex expressions, storing and printing out the results. To close the Python when you are done type `exit()` or `Ctrl-d`.

Creating Python script files

Python can be used interactively (one command at a time as we did above) or in "script" mode. In "script" mode, Python reads its commands from a file. Let's create a simple Python script that will calculate the amount of a radioactive substance that remains after a specific time given an initial amount of a substance, its half-life and the elapsed time. First, make sure you are in the scripts directory for this lab:

```
cd ~/lab1 [ENTER]
```

Then create a new text file with Atom as described before. Save this new blank file as `decay.py`. All of the Python scripts you create for this class will have the extension `.py` (in Linux, this doesn't really matter, but it's just convention)

After you have saved the blank file, you can enter the source code. Type in (or copy and paste) the following Python code:

```
init = float(input("Initial Amount: "))
halflife = float(input("Half-life: "))
time = float(input("Elapsed-time: "))
residual = init*0.5**(time/halflife)
print("Residual amount remaining = ",residual)
```

Make sure you type in the script *exactly* as shown! Note that computer languages are very precise and Python has no tolerance for errors! Every symbol in this program has significance and must be entered *exactly* as it appears.

Save your Python script as explained earlier.

Radioactive Decay

Your `decay.py` script should now exist as a file in the `lab1` directory. Change your working directory to the `lab1` directory that contains the `decay.py` script you created earlier. To run the script, simply add the file name argument (`decay.py`) to the `python3` command:

```
python3 decay.py[ENTER]
```

Try the following values and see how your script runs!

Initial amount = 4.
Half-Life = 140.
Elapsed Time = 420.

Turtle Graphics

Python has a built-in graphics display capability that is very simple and fun to use called "turtle" graphics. To use the turtle graphics package, you must first tell the Python interpreter to load the turtle *module*. Start up a new python interpreter and type the following Python commands (exactly as shown):

```
>>> import turtle [ENTER]
>>> turtle.showturtle() [ENTER]
```

The first Python command loads the turtle graphics module, the second one causes a graphics window to be displayed with a small triangle (the 'turtle') in the middle. To draw a line, you simply *move* the turtle. Try this:

```
>>> turtle.forward(100) [ENTER]
```

Voila! You've drawn a line in the direction the turtle is facing. Its easy to change the turtle direction using a rotation method. Type in the following Python instructions:

```
>>> turtle.left(90) [ENTER]
```

```
>>> turtle.forward(100) [ENTER]
```

There are lots of fun things you can do with turtles! Let's complete the square. Type the following:

```
>>> turtle.forward(100) [ENTER]
```

Oops, forgot to *turn* the turtle first. No problem, there is a very useful turtle command that will fix our mistake. Try the following:

```
>>> turtle.undo() [ENTER]
```

So you can try things and if they don't do what you expect, no problem, just "undo" them!

Now enter commands to complete the square on your own.

More Fun With Turtles

https://docs.python.org/3/ will be a very helpful website throughout the course of the semester. This website is essentially the "Instruction Manual" for the Python Language.

The official Python turtle reference is available at this website:

https://docs.python.org/3/library/turtle.html

Check it out and try as many of the turtle commands as you have time for. Go ahead! Experiment! It's fun! You simply type "turtle", followed by a period ('.') and then the name of the turtle command (method). Be sure to use the parentheses.

Now do something creative and show one of the TAs.

When you are done experimenting and want to end your Python session, simply enter `Ctrl-d` (hold the "control" key down and press the 'd' key at the same time). If you need help, ask a class TA for assistance.

If you've done this successfully, you should see your Unix command-line prompt once again.

Congratulations! You have completed your first lab exercise and are now an experienced Python programmer!