

Lab - Learn about Python

Objectives

Part 1: Basic Linux Commands

Part 2: Using the Python Shell

Part 3: Writing a Script in Python

Part 4: Creating a Simple Ping Sweeper with Python

Scenario

Python is a high-level, interpreted programming/scripting language created in 1989 by Guido van Rossum. Python is included by default as a component in the Linux and OSX operating systems, and can be installed on Windows. Python is the main user programming language for the Raspberry Pi. Python has a very large standard code library. Compared to other programming languages, Python code is uncluttered and easily readable.

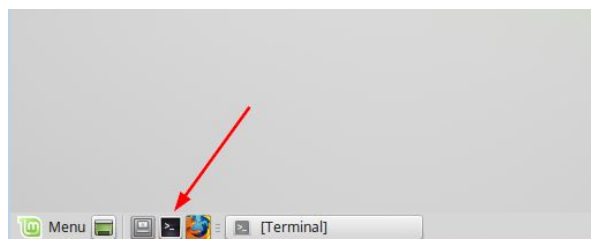
In this lab, you will use a Linux Mint virtual machine that you created previously for Python programming.

Part 1: Basic Linux Commands

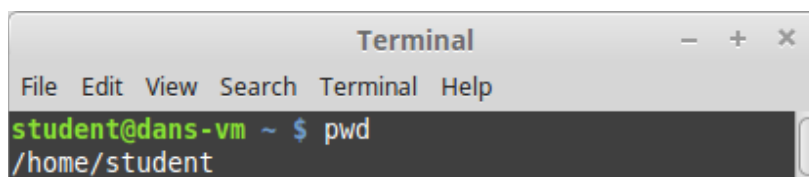
This part contains an introduction to some basic Linux commands, such as `pwd`, `cd`, and `mv`. If you know how to navigate through the directories, create directories, and manage files, you can skip this part.

When you open a terminal window, the prompt indicates where you are currently in the directory tree.

- In Linux Mint, click the terminal shortcut at the bottom of the window near the Menu to open a terminal.



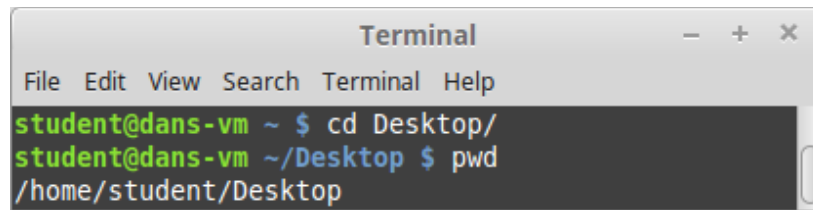
- Type **`pwd`** at the prompt.



What is the current directory?

- To navigate to your Desktop (`home/student/Desktop`), type **`cd Desktop`** at the prompt. To verify your current directory, type **`pwd`** at the prompt.

What is the current directory?



```
Terminal
File Edit View Search Terminal Help
student@dans-vm ~ $ cd Desktop/
student@dans-vm ~/Desktop $ pwd
/home/student/Desktop
```

Another way to determine your location in the directory tree is to look at the prompt. In this example, the prompt, **student@dans-vm ~/Desktop \$**, provides the name of the current user, the computer name, the current working directory, and the privilege level.



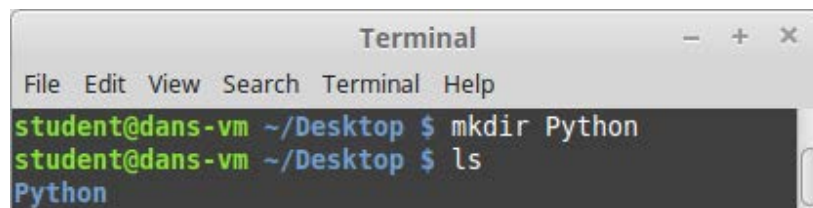
~/Desktop: is the current working directory. The symbol **~** represents the current user's home directory. In this example, it is **/home/student**.

\$: indicates regular user privilege. If **#** is displayed at the prompt, this indicates elevated privilege (root).

- d. At the prompt, type **ls** to list what is in your current directory.

Note: There may not be any files or directories in your current directory.

- e. To create a new directory named **Python** in your current directory, type **mkdir Python** at the prompt. To verify the creation of the new folder, type **ls** at the prompt.



```
Terminal
File Edit View Search Terminal Help
student@dans-vm ~/Desktop $ mkdir Python
student@dans-vm ~/Desktop $ ls
Python
```

- f. You can create a text file in your current directory and move it to a different directory. At the prompt, type **touch myscript.py** to create an empty file named **myscript.py**.

```
student@dans-vm ~/Desktop $ touch myscript.py
```

Type **ls** to verify that the file has been created. In what directory is the new file located?

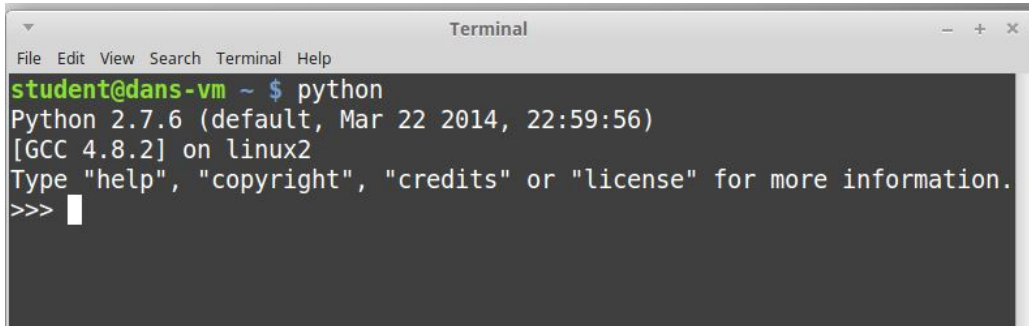
- g. The **mv** command is used to move or rename a file. To move **myscript.py** to the home directory, type **mv myscript.py /home/student**.
- h. Type **ls** to verify the file has been moved. Type **ls /home/student** to verify that the file has been moved to the student's home directory.
- i. To learn more about a particular command using the online reference manual, type **man command** at the prompt. Using **ls** as an example, type **man ls** at the prompt.

Part 2: Using the Python Shell

In Part 2, you will practice using the Python interpreter/shell.

Step 1: Use the terminal to run Python.

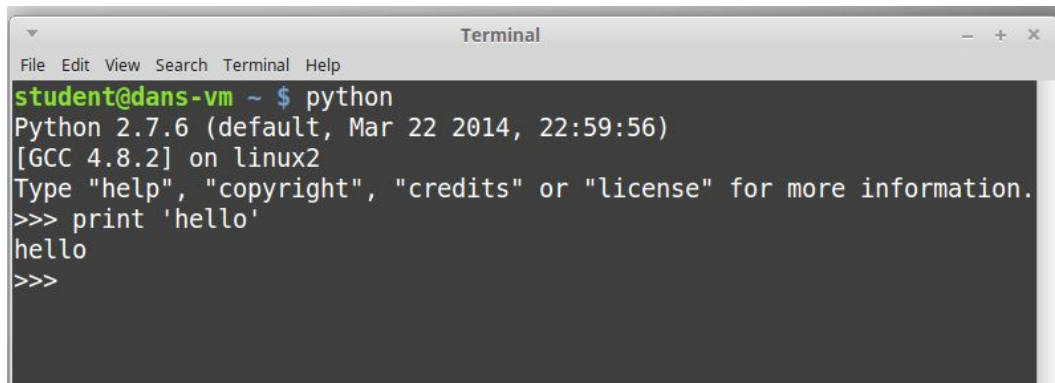
- Python is installed on most Linux operating systems by default. To start the Python interpreter on your Linux virtual machine, open a terminal shell.
- In the terminal, type **python** to launch Python. You should see the default version of Python and the Python shell command prompt **>>>**. The image below shows the preinstalled version of Python is 2.7.6. There are newer releases of Python but to explore the basics, this default version is fine.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command `student@dans-vm ~ $ python` being executed. The output is: `Python 2.7.6 (default, Mar 22 2014, 22:59:56)`, `[GCC 4.8.2] on linux2`, and `Type "help", "copyright", "credits" or "license" for more information.` followed by the prompt `>>>` and a cursor.

Note: The terminal font size can be modified by going to **Edit > Profile Preferences** and changing the font size under the General tab. The background transparency can be changed by clicking on the Background tab and choosing solid color. The image above reflects these minor changes to the terminal window for readability.

Step 2: Say hello with Python.

To say hello with Python type **print 'hello'** where the string of text is nested between double or single quotes and press Enter.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command `student@dans-vm ~ $ python` being executed. The output is: `Python 2.7.6 (default, Mar 22 2014, 22:59:56)`, `[GCC 4.8.2] on linux2`, and `Type "help", "copyright", "credits" or "license" for more information.` followed by the prompt `>>>`. The user enters `print 'hello'`, and the output is `hello`, followed by the prompt `>>>`.

Step 3: Create variables.

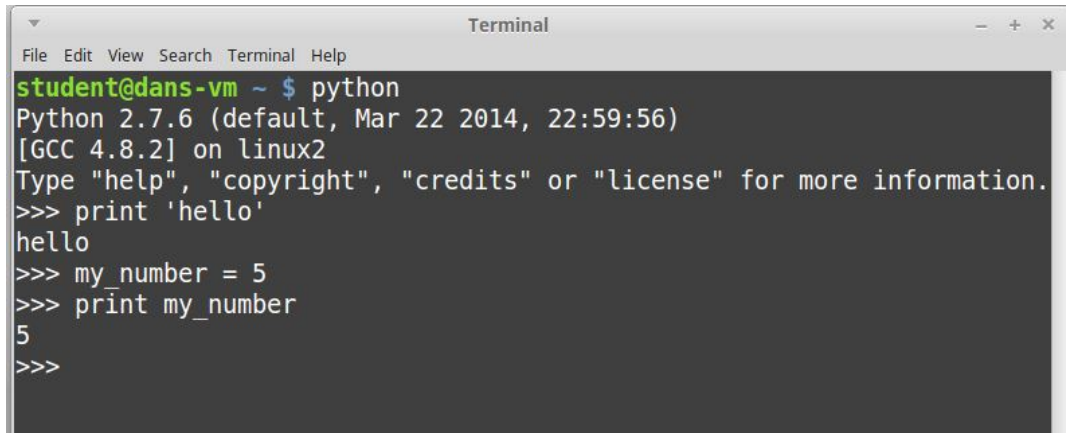
In programming, variables are labelled containers that are created by the programmer to hold values that will be needed in the program. Variables are typically set to equal numbers (integers and floating points - computer representation of decimal points), text (strings), or booleans (values for true or false). Variables can also be set to other values like arrays (lists) and functions. In Python, variable names are case sensitive and can be made up of letters, numbers and underscores, but must start with a letter and cannot have any spaces.

- To create a variable and assign it a value, use an allowed name for the variable, followed by an equal sign and then the numeric value. Create a variable called `my_number` and assign it a number or integer value of 5:

```
>>> my_number = 5
```

- b. Display the content of the variable by using the **print my_number** command:

```
>>> print my_number
5
```

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is "student@dans-vm ~ \$". The user enters "python", which starts the Python 2.7.6 interpreter. The prompt changes to ">>>". The user enters "print 'hello'", which outputs "hello". Then the user enters "my_number = 5", followed by "print my_number", which outputs "5". The prompt returns to ">>>".

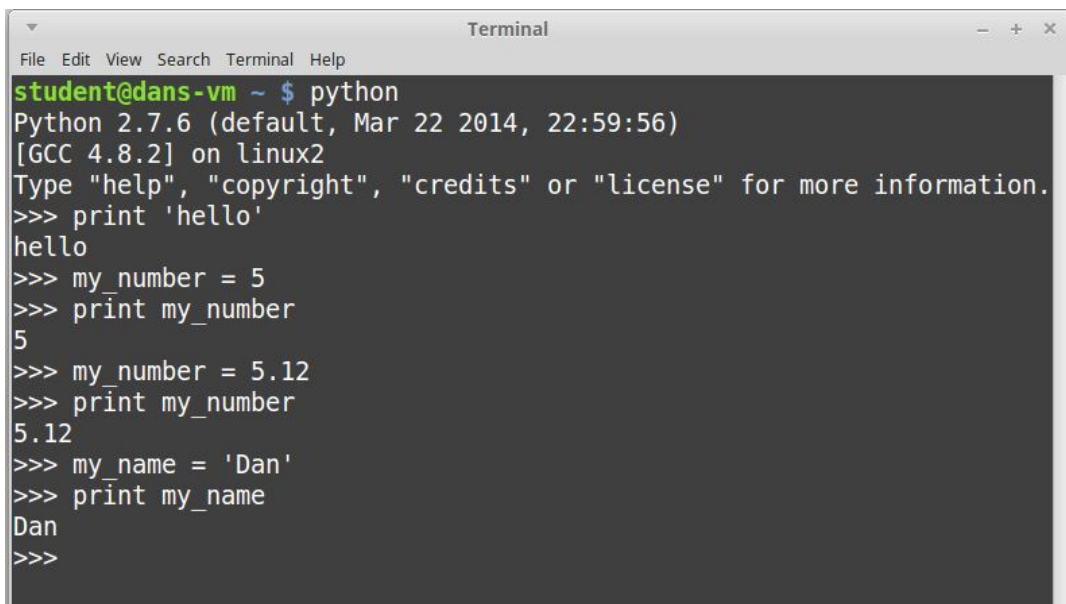
```
student@dans-vm ~ $ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello'
hello
>>> my_number = 5
>>> print my_number
5
>>>
```

- c. To change a variable's value, assign it a new value with the equal sign. Change `my_number` to a floating point value:

```
>>> my_number = 5.12
>>> print my_number
5.12
```

- d. Similar to a number variable, create a variable and set it to equal a string of text. Use an allowed name for the variable, followed by an equal sign, and then a string of text enclosed within single or double quotes. Create a variable named `my_name` and set it to equal your name.

```
>>> my_name = 'Dan'
>>> print my_name
Dan
```

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is "student@dans-vm ~ \$". The user enters "python", which starts the Python 2.7.6 interpreter. The prompt changes to ">>>". The user enters "print 'hello'", which outputs "hello". Then the user enters "my_number = 5", followed by "print my_number", which outputs "5". Next, the user enters "my_number = 5.12", followed by "print my_number", which outputs "5.12". Finally, the user enters "my_name = 'Dan'", followed by "print my_name", which outputs "Dan". The prompt returns to ">>>".

```
student@dans-vm ~ $ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello'
hello
>>> my_number = 5
>>> print my_number
5
>>> my_number = 5.12
>>> print my_number
5.12
>>> my_name = 'Dan'
>>> print my_name
Dan
>>>
```

- e. To create a Boolean variable in Python, set the variable equal to True or False with an initial capital letter T or F. (Using all lower case letters will not work.) You can indirectly create a Boolean variable by setting the variable equal to 0, or an empty string "". In Python, any object can be tested for its truth value. The following examples of the game_over variable are all equal to False.

Set **game_over** to **False**:

```
>>> game_over = False
>>> print game_over
False
```

Set **game_over** to **0**:

```
>>> game_over = 0
>>> print game_over
0
```

Set **game_over** to **""**:

```
>>> game_over = ''
>>> print game_over
```

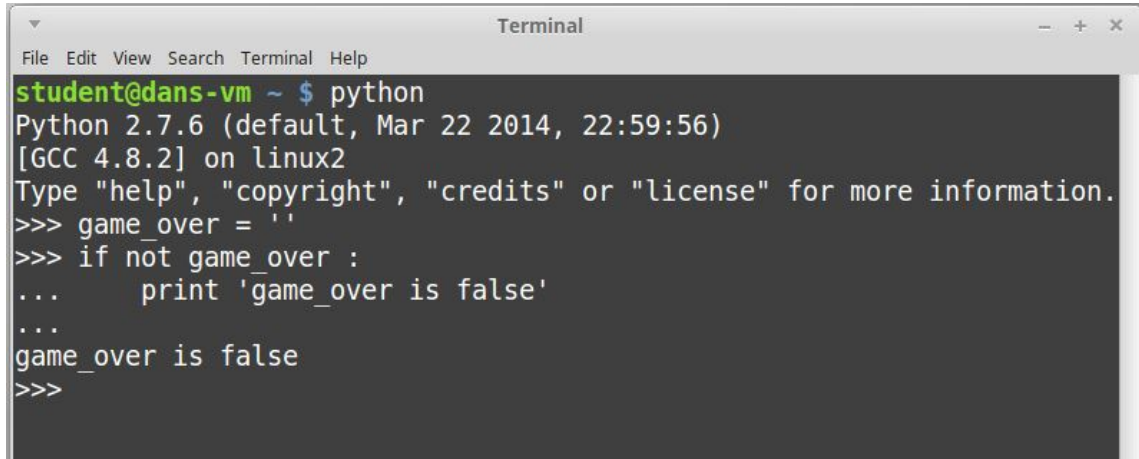
Notice in the last example that when game_over is set to an empty string and printed to the console it returns nothing. This is a good time to start using conditional statements to test for true or false.

Step 4: Flow control with conditional statements **if**, **else if**, and **else**.

Like entering commands in the console one line at a time, programs will execute instructions sequentially. However, conditional statements can change the flow of the program by executing different blocks of instructions. Flow control in a program can be based on conditions using comparison tests and logic. The most basic way to control the flow of a program is by using if statements.

- a. In the previous step, you set the variable game_over to an empty string "" which is the equivalent of a False. Using an if statement, you can run a test to see if the variable is equal to False. To test if game_over is equal to False, use the **if** command, followed by the **not** keyword, the object you are testing, and end the line with a **colon**. Press **Enter**. On the next line, press the **tab** key once and type print "game_over is false". Press enter twice to execute the statement.

```
>>> game_over = ''
>>> if not game_over :
... <press tab key>print 'game_over is false'
...
game_over is false
```

A screenshot of a terminal window titled "Terminal". The window shows the execution of a Python script. The prompt is "student@dans-vm ~ \$". The command "python" is entered, followed by the Python version "Python 2.7.6 (default, Mar 22 2014, 22:59:56)" and the platform "[GCC 4.8.2] on linux2". The user is prompted to type "help", "copyright", "credits" or "license" for more information. The code being executed is the same as in the previous block: ">>> game_over = ''", ">>> if not game_over :", "... <press tab key>print 'game_over is false'", "...", and "game_over is false". The prompt ">>>" appears again at the end of the output.

```
student@dans-vm ~ $ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> game_over = ''
>>> if not game_over :
...     print 'game_over is false'
...
game_over is false
>>>
```

The string of text was printed to the screen because the testing of if `game_over` is false, returns true. It was a true statement that `game_over` is set to false.

Try the commands above again, except this time make `game_over` equal to 0. Does '`game_over` is false' print to the screen?

- b. Notice that the command prompt after the line ending with a colon shows three dots `...`. This indicates that additional code is required in the if statement. Python does not require parenthesis and curly braces within its code blocks like many other languages; however it does require indentation in order to function correctly. Try running the previous if statement without the tab spacing and you will see the error output.

```
>>> if not game_over :
... print 'game_over is false'
      File "<stdin>", line 2
        print "game_over is false"
            ^
IndentationError: expected an indented block
>>>
```

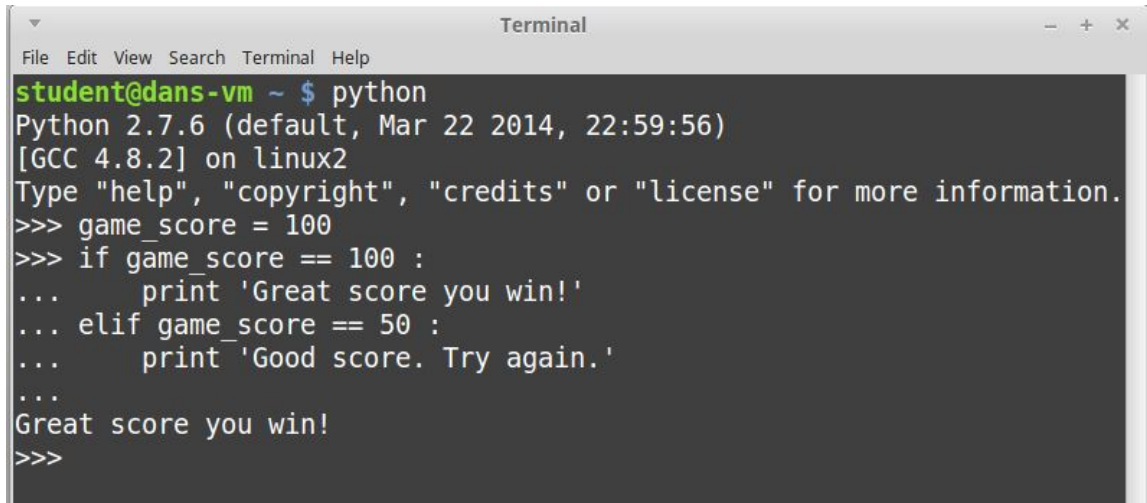
- c. To see the if statement return False and not execute the print statement, run a test to see if `game_over` is True instead of False. To do this, repeat the if statement, but remove the keyword **not**. Use a **tab** before the print statement, and press enter twice.

```
>>> if game_over :
... <press tab key>print 'game_over is false'
...
>>>
```

The if statement is a test to see if the variable `game_over` is True. Because the variable is not equal to True, the if statement returns False, and the print statement is not executed.

- d. To use **if** and **else if** statements together, create a variable named `game_score` and set it equal to 100. Set up the following conditional tests using **if** and **elif** statements (**elif** stands for else if.). The double equal sign is used as a comparison test of equality.

```
>>> game_score = 100
>>> if game_score == 100 :
... <press tab key>print 'Great score you win!'
... elif game_score == 50 :
... <press tab key>print 'Good score. Try again.'
...
Great score you win!
```



```
Terminal
File Edit View Search Terminal Help
student@dans-vm ~ $ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> game_score = 100
>>> if game_score == 100 :
...     print 'Great score you win!'
... elif game_score == 50 :
...     print 'Good score. Try again.'
...
Great score you win!
>>>
```

Try it again but change `game_score` to 50. What was printed to screen?

- e. Use the **else** condition at the end of the conditional block of code. The following code also uses less-than or equal comparison operators.

```
>>> game_score = 101
>>> if game_score <= 50 :
... <press tab key>print 'Level 1'
... elif game_score <= 100 :
... <press tab key>print 'Level 2'
... else :
... <press tab key>print 'Level 3'
...
Level 3
>>>
```

Part 3: Writing a Script in Python

In Part 3 you will practice writing a script and running it with the Python interpreter.

Step 1: Use a text editor and a for loop to create a Python script.

To write a script in Python, you typically use either a text editor or an integrated development environment (IDE). In this lab, you will use the program nano, a simple command line text editor to create Python script files.

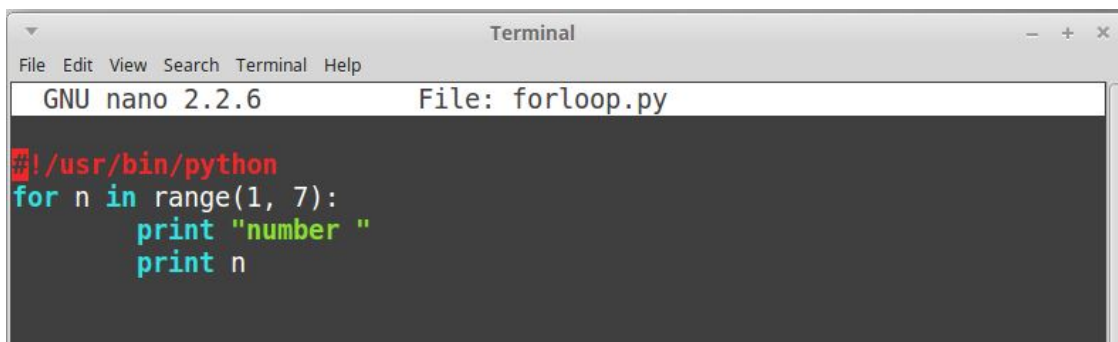
- Exit the Python console by pressing **Ctrl+D**.
- In the terminal, create a text file with nano by typing nano followed by the file name.

```
student@dans-vm ~ $ nano forloop.py
```

Linux does not require a file extension like Windows does in order to work correctly. However, naming the file with .py at the end of the file name helps to make the file easily recognizable as a Python script.

- In the nano program, type the following lines of text. The first line is the path to the Python interpreter. The second line is a for loop which is used when you want to repeat lines of code a specific number of times. In the script, the for loop repeats 6 times, where the variable **n** starting at number 1, increments 6 times ending at number 6. The print 'number and print n statements repeat and execute 6 times.

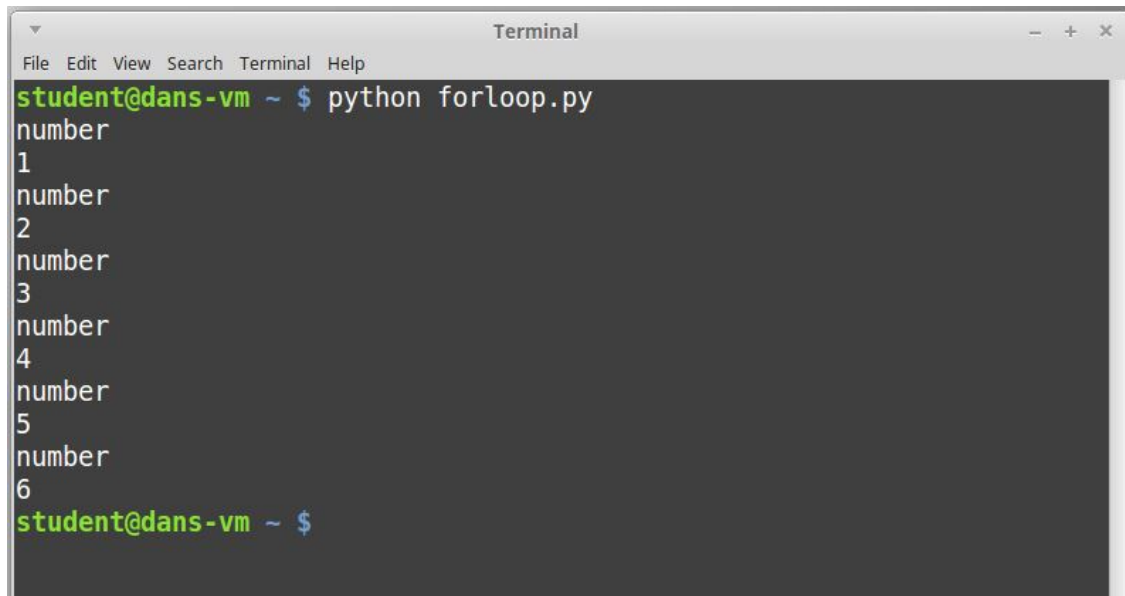
```
#!/usr/bin/python
for n in range(1, 7):
    <press tab key>print 'number '
    <press tab key>print n
```



To save the file, press **Ctrl+X** to exit. When asked if you want to save the file, press **y** for yes, and then press enter to accept the file name.

- d. To run the script, type python followed by the file path to the script you want to run.

```
student@dans-vm ~ $ python forloop.py
```

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command `python forloop.py` being executed. The output consists of six lines, each starting with "number" followed by a number from 1 to 6. The prompt `student@dans-vm ~ $` is visible at the bottom of the terminal.

```
student@dans-vm ~ $ python forloop.py
number
1
number
2
number
3
number
4
number
5
number
6
student@dans-vm ~ $
```

Note: If you received any error, edit the file again by entering **nano forloop.py** at the prompt.

Step 2: Create an executable script with file permissions.

- a. If you want to run the `forloop.py` script without having to type `python` before the path to the file name, you need to add the execute permission to the file's permissions. To add the execute permission to `forloop.py` run the following command:

```
student@dans-vm ~ $ chmod +x forloop.py
```

- b. Because the file now has execute permissions, you can run the file by simply typing the file name prefaced with the relative current directory file path (`./`).

```
student@dans-vm ~ $ ./forloop.py
number
1
number
2
number
3
number
4
number
5
number
6
```

Part 4: Creating a Simple Ping Sweeper with Python

A ping sweeper is a program that can ping an entire network or range of IP addresses.

Step 1: Use a text editor to create the script.

- a. In the terminal, use nano to create the file pingsweeper.py:

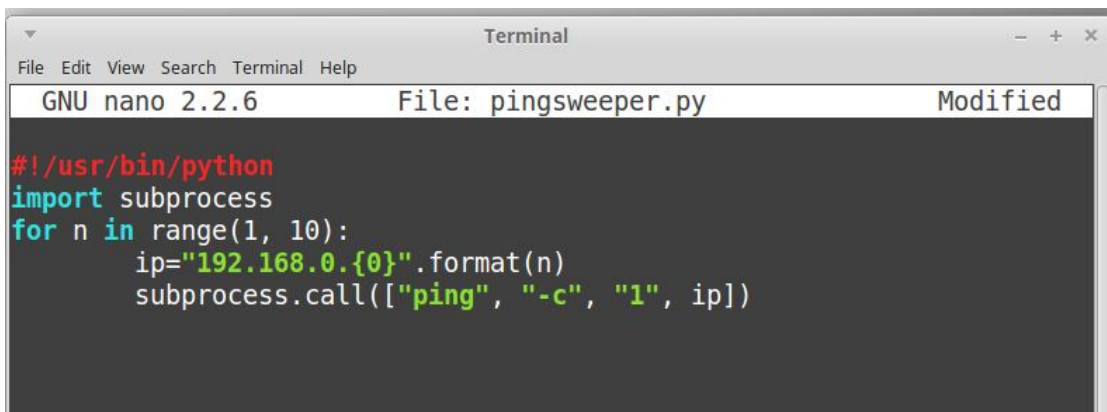
```
student@dans-vm ~ $ nano pingsweeper.py
```

- b. Make the file executable by adding the execute permission

```
student@dans-vm ~ $ chmod +x pingsweeper.py
```

- c. In nano, type the following lines of text.

```
#!/usr/bin/python
import subprocess
for n in range(1, 10) :
<press tab key>ip="192.168.0.{0}".format(n)
<press tab key>subprocess.call(["ping", "-c", "1", ip])
```



In the above script, the lines of code do the following:

- 1) Line 1 is the path to the Python interpreter.
 - 2) Line 2 imports the subprocess code library or module into the program.
 - 3) Line 3 is a for loop which loops through numbers 1 to 9, where n is the number.
 - 4) Line 4 sets the variable ip to a string of text with a replaceable field {0} that is replaced with the value of n. You should replace the network portion of the ip address 192.168.0. with your own network's address.
 - 5) Line 5 opens a subprocess by executing pings. The pings have a count of 1 so they ping the ip address in the ip variable 1 time each.
- d. Save the script by pressing **Ctrl+X**, typing **Y**, and pressing **Enter**.
- e. Make the file executable by adding the execute permission:
- ```
student@dans-vm ~ $ chmod +x pingsweeper.py
```

- f. Run the script.

```
student@dans-vm ~ $./pingsweeper.py
```

```
student@dans-vm ~ $./pingsweeper.py
PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=1004 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1004.519/1004.519/1004.519/0.000 ms
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=16.5 ms

--- 192.168.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 16.528/16.528/16.528/0.000 ms
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=1012 ms
```

Scroll through the command output to see the results of the 9 pings.

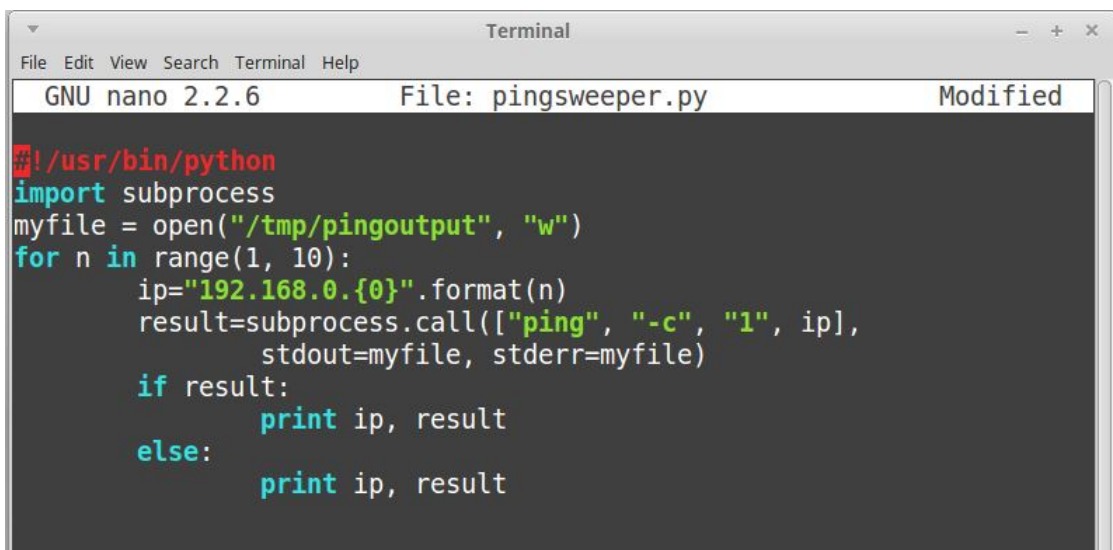
### Step 2: Improve the pingsweeper.py script by adding conditional logic.

- a. Open the pingsweeper.py script in nano.

```
student@dans-vm ~ $ nano pingsweeper.py
```

- b. In nano, make the following changes (shown in bold below).

```
#!/usr/bin/python
import subprocess
myfile = open("/tmp/pingoutput", "w")
for n in range(1, 10) :
 <press tab key>ip="192.168.0.{0}".format(n)
 <press tab key>result = subprocess.call(["ping", "-c", "1", ip],
 <press tab key><press tab key>stdout=myfile, stderr=myfile)
 <press tab key>if result :
 <press tab key><press tab key>print ip, result
 <press tab key>else :
 <press tab key><press tab key>print ip result
```



In the above script, the added lines of code do the following:

- 1) Line 3, creates a file named myfile in the temporary files directory.
  - 2) Line 6, each ping subprocess is loaded into a variable name result and the output is sent to myfile in the temporary files directory.
  - 3) Lines 8-11, based on whether the value of result is True or False, the if else conditional statements print the pinged IP addresses to the screen along with the value of the result.
- c. Save and exit the script by pressing **Ctrl+X**, typing **Y**, and pressing **Enter**.
- d. Run the script. For the IP addresses that responded successfully to pings the result variable returned 0s, and for the IP addresses that did not respond to pings the result variable returned 1s.

```
student@dans-vm ~ $./pingsweeper.py
192.168.0.1 0
192.168.0.2 0
192.168.0.3 0
192.168.0.4 1
192.168.0.5 1
192.168.0.6 1
192.168.0.7 1
```

```
192.168.0.8 1
```

```
192.168.0.9 1
```

- e. Open the pingsweeper.py script in nano.

```
student@dans-vm ~ $ nano pingsweeper.py
```

- f. Make the following changes (shown in bold below).

```
#!/usr/bin/python
import subprocess
myfile = open("/tmp/pingoutput", "w")
for n in range(1, 10) :
 <press tab key>ip="192.168.0.{0}".format(n)
 <press tab key>result = subprocess.call(["ping", "-c", "1", ip],
 <press tab key><press tab key>stdout=myfile, stderr=myfile)
 <press tab key>if result :
 <press tab key><press tab key>print ip, 'is off'
 <press tab key>else :
 <press tab key><press tab key>print ip, 'is on'
```

- g. Save and exit the script by pressing **Ctrl+X**, typing **Y**, and pressing **Enter**.

- h. Run the script, and now the hosts that respond to pings should be easily recognizable.

```
student@dans-vm ~ $./pingsweeper.py
```

```
192.168.0.1 is on
```

```
192.168.0.2 is on
```

```
192.168.0.3 is on
```

```
192.168.0.4 is off
```

```
192.168.0.5 is off
```

```
192.168.0.6 is off
```

```
192.168.0.7 is off
```

```
192.168.0.8 is off
```

```
192.168.0.9 is off
```

If you could, what small programs would you write in Python to automate or improve your daily work flow?