

Lab - Construct a Basic Python Script (Instructor Version)

Instructor Note: Red font color or gray highlights indicate text that appears in the instructor copy only.

Answers: [28.1.2 Lab - Construct a Basic Python Script](#)

Objectives

Part 1: Explore the Python Interpreter

Part 2: Explore Data Types, Variables, and Conversions

Part 3: Explore Lists and Dictionaries

Part 4: Explore User Input

Part 5: Explore If Functions and Loops

Part 6: Explore File Access

Background / Scenario

In this lab, you will learn basic programming concepts using the Python programming language. You will create a variety of Python scripts. The lab culminates in a challenge activity in which you use many of the skills learned in this lab to construct a Python script.

Required Resources

- 1 PC (Choice of operating system with Cisco Networking Academy CCNP in a virtual machine client)
- Internet access (Optional)

Instructions

Part 1: Explore the Python Interpreter

In Part 1, you will start Python 3, use the interactive interpreter, open IDLE, and use IDLE to run your first script.

Step 1: Start the CCNP VM.

Note: If you have not completed **Lab - Install the CCNP Virtual Machine**, do so now before continuing with this lab.

- Open VirtualBox and start the **CCNP VM** virtual machine.
- Enter the password **StudentPass** to log in if prompted.

Step 2: Start Python 3.

- To start Python, open a terminal window and type **python3**. The three angle brackets (>>>) indicate that you are in Python's interactive interpreter.

```
student@CCNP:~$ python3
Python 3.6.9 (default, Nov 7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

Step 3: Use the interactive interpreter as a calculator.

From here, you can do a variety of basic programming tasks including math operations. The table shows the Python syntax to use for the most common math operators.

Operation	Math	Syntax
Addition	$a+b$	<code>a+b</code>
Subtraction	$a-b$	<code>a-b</code>
Multiplication	$a \times b$	<code>a*b</code>
Division	$a \div b$	<code>a/b</code>
Exponents	a^b	<code>a**b</code>

Enter some math operations using the Python syntax, as shown below.

```
>>> 2+3
5
>>> 10-4
6
>>> 2*4
8
>>> 20/5
4.0
>>> 3**2
9
```

Step 4: Use the interpreter to print a string.

A string is any sequence of characters such as letters, numbers, symbols, or punctuation marks. The interactive interpreter will directly output text that you enter as a string provided you enclose the string in either single quotes (') or double quotes (").

The **print** command can be used in a script to output a string. Enter the following in the interpreter:

```
>>> "Hello World!"
'Hello World!'
>>> 'Hello World!'
'Hello World!'
>>> print("Hello World!")
Hello World!
```

Step 5: Open the IDLE interactive interpreter.

There are many development environments available for programmers to manage their coding projects. However, the labs in this course use Python's Integrated Development Environment (IDLE).

To access IDLE on the **CCNP VM**, enter **quit()** in the interactive interpreter to exit and then type **idle** at the command line to launch IDLE.

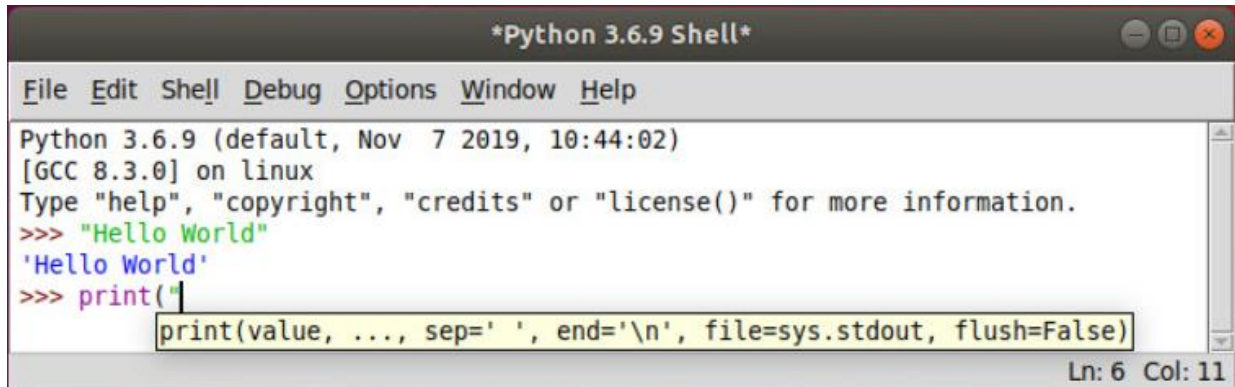
```
>>> quit()
student@CCNP:~$ idle
```

Lab - Construct a Basic Python Script

IDLE has two main windows:

- IDLE Shell
- IDLE Editor

The **IDLE Shell** provides an interactive interpreter with colorizing of code input, output, and error messages. It also includes a popup that provides syntactical help for the command you are currently using, as shown in the figure.



```
*Python 3.6.9 Shell*
File Edit Shell Debug Options Window Help
Python 3.6.9 (default, Nov 7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> "Hello World"
'Hello World'
>>> print(
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

The **IDLE Editor** provides a text editor with code colorization and syntactical help for writing python scripts. To open the IDLE Editor, in the IDLE Shell, click **File > New File**, as shown in the figure.

The **IDLE Editor** includes the ability to immediately test a script in the shell by using the Run Module (F5) command, as shown in the figure.

Step 6: Use IDLE to write, save, and run your first program.

Complete the following steps in IDLE:

- a. In IDLE shell, click **File > New File (Ctrl+N)** to open an Untitled script file.
- b. Save the file as **01_hello-world.py**.
- c. Enter the following in the script: **print("Hello World!")**.
- d. Save the script; click **File > Save (Ctrl+S)**.
- e. Run the script; click **Run > Run Module (F5)**.

Part 2: Explore Data Types, Variables, and Conversions

In Part 2, you will learn about different data types, work with variables, and convert one data type to another data type.

Step 1: Determine the basic data types.

In programming, data types are a classification which tells the interpreter how the programmer intends to use the data. For example, the interpreter needs to know if the data the programmer entered is a number or a string. Although there are several different data types, we will focus only on the following:

- **Integer** - used to specify whole numbers (no decimals), such as 1, 2, 3, etc.. If an integer is entered with a decimal, the interpreter ignores the decimal. For example, 3.75 is interpreted as 3.
- **Float** - used to specify numbers that need a decimal value, such as 3.14159.
- **String** - any sequence of characters such as letters, numbers, symbols, or punctuation marks.
- **Boolean** - any data type that has a value of either "True" or "False".

Lab - Construct a Basic Python Script

Use the `type()` function to determine the data type.

```
>>> type(98)
<class 'int'>
>>> type(98.6)
<class 'float'>
>>> type("Hi!")
<class 'str'>
>>> type(True)
<class 'bool'>
```

Step 2: Use Boolean operators.

The Boolean data type makes use of the operators shown in the table.

Operator	Meaning
>	Greater than
<	Less than
==	Equal to
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

In the IDLE shell, try out the different Boolean operators. The following are a few examples.

```
>>> 1<2
True
>>> 1>2
False
>>> 1==1
True
>>> 1!=1
False
>>> 1>=1
True
>>> 1<=1
True
```

Step 3: Create and use a variable.

The Boolean operator for determining whether two values are equal is the double equal sign (`==`). A single equal sign (`=`) is used to assign a value to a variable. The variable can then be used in other commands to recall the assigned value, as shown in the following example. Try creating and using your own variables.

```
>>> x=3
>>> x*5
15
>>> "Cisco"*x
```

```
'CiscoCiscoCisco'
```

Step 4: Concatenate multiple string variables.

Concatenation is the process of combining multiple strings into one string. For example, the concatenation of "foot" and "ball" is "football". In the following example, four variables are concatenated together in a `print()` statement with the plus sign (+). Notice that the space variable was defined for use as white space between the words.

```
>>> str1="Cisco"
>>> str2="Networking"

>>> str3="Academy"
>>> space=" "
>>> print(str1+space+str2+space+str3)
Cisco Networking Academy
```

Challenge: Try writing a `print()` statement with a space between the words without using a variable to create the space.

```
>>> print(str1+" "+str2+" "+str3)
Cisco Networking Academy
```

Step 5: Convert one data type to another data type.

- a. Concatenation does not work for joining different data types, as shown in the following example:

```
>>> x=3
>>> print("The value of X is " + x)
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print("The value of X is " + x)
TypeError: must be str, not int
```

- b. Use the `str()` function to convert the value of a variable to a string, as shown in the following example:

```
>>> print("The value of x is " + str(x))
The value of x is 3
>>> type(x)
<class 'int'>
```

- c. Notice that the data type for the variable x is still an integer, as confirmed using the `type()` function. To convert the data type, reassign the variable to the new data type, as shown in following example:

```
>>> x=str(x)
>>> type(x)
<class 'str'>
```

- d. You may want to display a float to a specific number of decimal places instead of the full number. To do this, use the `"{: .2f} ".format` function, as shown in the following example:

```
>>> num = 22/7
>>> print(num)
3.142857142857143
>>> print("{: .2f} ".format(num))
3.14
```

Part 3: Explore Lists and Dictionaries

In Part 3, you will work with list variables and dictionaries. You will then troubleshoot a program that has some errors.

Step 1: Create, use, and modify a list variable.

- a. In programming, a list variable is used to store multiple pieces of ordered information. The following example shows how to create a list and then use the **type()** and **len()** functions.

- 1) Create a list using brackets [] and enclosing each item in the list with quotes.
- 2) Separate the items with a comma.
- 3) Use the **type()** command to verify the data type.
- 4) Use the **len()** command return the number of items in a list.
- 5) Call the list variable name to display the list contents.

```
>>> hostnames=["R1", "R2", "R3", "S1", "S2"]
>>> type(hostnames)
<class 'list'>
>>> len(hostnames)
5
>>> hostnames
['R1', 'R2', 'R3', 'S1', 'S2']
```

- b. In some programming environments, an item list is also called an array. Any item within a list can be referenced and manipulated using its index, as shown below.

- 1) The first item in a list is indexed as zero, the second is indexed as one, and so on.
- 2) The last item can be referenced with index [-1].
- 3) Replace an item by assigning a new value to the index.
- 4) Use the **del** command to remove an item from a list.

```
>>> hostnames[0]
'R1'
>>> hostnames[-1]
'S2'
>>> hostnames[0]="RTR1"
>>> hostnames
['RTR1', 'R2', 'R3', 'S1', 'S2']
>>> del hostnames[3]
>>> hostnames
['RTR1', 'R2', 'R3', 'S2']
```

Step 2: Create, use, and modify a dictionary.

- a. Dictionaries are unordered lists of objects. Each object contains a key/value pair. In the following example, a dictionary **ipAddress** is created with three key/value pairs to specify the IP address values for three routers.

- 1) Create a dictionary using the braces { }.
- 2) Each dictionary entry includes a key and a value.

Lab - Construct a Basic Python Script

- 3) Separate a key and its value with a colon.
- 4) Use quotes for keys and values that are strings.

```
>>> ipAddress = {"R1": "10.1.1.1", "R2": "10.2.2.1", "R3": "10.3.3.1"}
>>> type(ipAddress)
<class 'dict'>
```

- b. Unlike list items, objects inside a dictionary cannot be referenced by their sequence number. Instead, you reference a dictionary object using its key, as shown in the following example:

- 1) The key is enclosed with brackets [].
- 2) Keys that are strings can be referenced using single or double quotes, as shown for R1 and S1 in Example 2.
- 3) Add a key/value pair by setting the new key equal to a value.
- 4) Use a **key in dictionary** statement to verify if a key exists in the dictionary.

```
>>> ipAddress
{'R1': '10.1.1.1', 'R2': '10.2.2.1', 'R3': '10.3.3.1'}
>>> ipAddress["R1"]
'10.1.1.1'
>>> ipAddress["S1"]="10.1.1.10"
>>> ipAddress
{'R1': '10.1.1.1', 'R2': '10.2.2.1', 'R3': '10.3.3.1', 'S1': '10.1.1.10'}
>>> "R3" in ipAddress
True
```

- c. Values in a key/value pair can be any other data type including lists and dictionaries. For example, if R3 has more than one IP address, how would you represent that inside the **ipAddress** dictionary? Create a list for the value of the R3 key, as shown in the following example:

```
>>> ipAddress["R3"]=["10.3.3.1", "10.3.3.2", "10.3.3.3"]
>>> ipAddress
{'S1': '10.1.1.10', 'R2': '10.2.2.1', 'R1': '10.1.1.1', 'R3': ['10.3.3.1', '10.3.3.2', '10.3.3.3']}
```

Step 3: Challenge: Troubleshoot a list and dictionary program.

The script creates a list of the BRICS countries (Brazil, Russia, India, China, and South Africa). It then creates a dictionary for each country's capital(s). A list is used for South Africa's three capitals. The script is then supposed to print the country list, capital dictionary, and the 2nd listed capital for the value of South Africa. However, there are errors in the script.

- a. Open a new file in the IDLE editor and save it as **02_list-dict.py**. Enter the following code into the file.

```
country=["Brazil", "Russia", "India", "China", "South Africa"]
capitals={"Brazil": "Brasilia", "Russia": "Moscow", "India": "New
Delhi", "China": "Beijing", "South Africa": ["Pretoria", "Cape
Town", "Bloemfontein"]}
print("country")
print("capitals")
print(capitals["South Africa"][1])
```

- b. Save **02_list-dict.py** and run the script. Each print statement has an error. Troubleshoot the print statements to resolve the issues. The output of a corrected script should look like the following:

Lab - Construct a Basic Python Script

```
===== RESTART: /home/Student/02_list-dicts.py =====
['Brazil', 'Russia', 'India', 'China', 'South Africa']
{'South Africa': ['Pretoria', 'Cape Town', 'Bloemfontein'], 'India': 'New Delhi',
 'Russia': 'Moscow', 'China': 'Beijing', 'Brazil': 'Brasilia'}
Cape Town
```

The print statements each have syntactical errors. The following are the correct print statements.

```
print(country)
print(capitals)
print(capitals["South Africa"][1])
```

Part 4: Explore User Input

In Part 4, you will create a program to ask for user input and use that information for data output.

Step 1: Create and use the input function.

Most programs require some type of input either from a database, another computer, mouse clicks, or keyboard input. For keyboard input, use the function `input()` which includes an optional parameter to provide a prompt string.

If the input function is called, the program will stop until the user provides input and hits the Enter key, as shown in the following example:

```
>>> firstName = input("What is your first name? ")
What is your first name? Bob
>>> print("Hello " + firstName + "!")
Hello Bob!
```

Step 2: Create a script to collect personal information.

- Open a blank script file and save it as `03_personal-info.py`.
- Create a script that asks for four pieces of information such as: first name, last name, location, and age.
- Create a variable for a space.
- Add a print statement that combines all the information in one sentence.
- Your script should run without any errors, as shown in the following output:

```
===== RESTART: /home/Student/03_personal-info.py =====
What is your first name? Bob
What is your last name? Smith
What is your location? London
What is your age? 36
Hi Bob Smith! Your location is London and you are 36 years old.
```

The following is one example of how to create the `03_personal-info.py` script.

```
firstName = input("What is your first name? ")
lastName = input("What is your last name? ")
location = input("What is your location? ")
age = input("What is your age? ")
space = " "
print("Hi " + firstName + space + lastName + "! Your location is " + location
+ " and you are " + age + " years old.")
```


Part 5: Explore If Functions and Loops

In programming, conditional statements check if something is true and then carry out instructions based on the evaluation. If the evaluation is false, different instructions may be carried out. In Part 5, you will explore conditional statements that include If/Else, If/Elif/Else, For loops, and While loops.

Step 1: Create an If/Else conditional statement.

The following example demonstrates the if/else function in a simple script:

```
nativeVLAN = 1
dataVLAN = 100
if nativeVLAN == dataVLAN:
    print("The native VLAN and the data VLAN are the same.")
else:
    print("The native VLAN and the data VLAN are different.")
```

a. Create this script for your files:

- 1) Open a blank script and save it as **04_if-vlan.py**.
- 2) Type the script above into your new file.
- 3) Run the script and troubleshoot any errors. Your output should look like the following example:

```
===== RESTART: /home/Student/04_if-vlan.py =====
The native VLAN and the data VLAN are different.
```

b. Modify the variables so that **nativeVLAN** and **dataVLAN** have the same value. Save and run the script again. Your output should look like the following example:

```
===== RESTART: /home/Student/04_if-vlan.py =====
The native VLAN and the data VLAN are the same.
```

Step 2: Create an If/Elif/Else conditional statement.

What if we have more than two conditional statements to consider? In this case, we can use **elif** statements in the middle of the **if/else** function. An **elif** statement is evaluated if the **if** statement is false and before the **else** statement. You can have as many **elif** statements as you would like. However, the first one matched will be executed and none of the remaining **elif** statements will be checked. Nor will the **else** statement.

The script in the following example asks the user to input the number of an IPv4 ACL and then checks whether that number is a standard IPv4 ACL, extended IPv4 ACL, or neither standard or extended IPv4 ACL.

Note: The data type for the input function is changed from the default string to an integer so that the **if** and **elif** evaluations will work.

```
aclNum = int(input("What is the IPv4 ACL number? "))
if aclNum >= 1 and aclNum <= 99:
    print("This is a standard IPv4 ACL.")
elif aclNum >=100 and aclNum <= 199:
    print("This is an extended IPv4 ACL.")
else:
    print("This is not a standard or extended IPv4 ACL.")
```

a. Create this script for your files:

- 1) Open a blank script and save it as **05_if-acl.py**.
- 2) Type the script above into your new file.

Lab - Construct a Basic Python Script

- b. Run multiple times to test each statement. Troubleshoot any errors. Your output should look similar to the following:

```
===== RESTART: /home/Student/05_if-acl.py =====
What is the IPv4 ACL number? 10
This is a standard IPv4 ACL.
>>>
===== RESTART: /home/Student/05_if-acl.py =====
What is the IPv4 ACL number? 100
This is an extended IPv4 ACL.
>>>
===== RESTART: /home/Student/05_if-acl.py =====
What is the IPv4 ACL number? 2000
This is not a standard or extended IPv4 ACL.
>>>
```

Step 3: Create a For loop.

The Python **for** command is used to loop or iterate through the elements in a list, or perform an operation on a series of values.

- a. Enter the following example in your interactive interpreter. The example demonstrates how a **for** loop can be used to print the elements in a list. The variable name **item** is arbitrary and can be anything the programmer chooses.

Note: After the last statement in a **for** loop, press the Enter key twice to end the **for** loop statement and run it.

```
>>> devices=["R1","R2","R3","S1","S2"]
>>> for item in devices:
    print(item)
```

```
R1
R2
R3
S1
S2
```

- b. What if you only want to list the items that begin with the letter R? An **if** statement can be embedded in a **for** loop to achieve this. Enter the following example in your interactive interpreter:

```
>>> for item in devices:
    if "R" in item:
        print(item)
```

```
R1
R2
R3
```

- c. You can also use a combination of the **for** loop and **if** statement to create a new list. The following shows how to use the **append()** method to create a new list called **switches**. In your interactive interpreter,

Lab - Construct a Basic Python Script

create an empty list variable for switches. Then iterate through **devices** to find and add any device that begin with an "S" to the **switches** list.

```
>>> switches=[]
>>> for item in devices:
    if "S" in item:
        switches.append(item)

>>> switches
['S1', 'S2']
```

Step 4: Create a While loop.

Instead of running a block of code once, as in an **if** statement, you can use a **while** loop. A **while** loop keeps executing a code block as long as a boolean expression remains true. This can cause a program to run endlessly if you do not make sure your script includes a condition for the **while** loop to stop. **While** loops will not stop until the boolean expression evaluates as false.

Note: If you need to interrupt an endless loop, you can press Ctrl-C, Ctrl-F6, or **Shell > Restart Shell** from the Shell.

a. Create this script for your files:

- 1) Open a blank script and save it as **06_while-loop.py**.
- 2) Create a program with a while loop that counts to a user's supplied number.
 - (i) Convert the string to an integer: **x = int(x)**.
 - (ii) Set a variable to start the count: **y = 1**.
 - (iii) While **y <= x**, print the value of y and increment y by 1.

```
x=input("Enter a number to count to: ")
x=int(x)
y=1
while y<=x:
    print(y)
    y=y+1
```

b. Run the script to get output similar to the following example:

```
===== RESTART: /home/Student/06_while-loop.py =====
Enter a number to count to: 10
1
2
3
4
5
6
7
8
9
10
```

- c. Instead of using **while y <= x**, we can modify the **while** loop to use a Boolean check and break to stop the loop when the check evaluates as false.

```
x=input("Enter a number to count to: ")
x=int(x)
y=1
while True:
    print(y)
    y=y+1
    if y>x:
        break
```

Modify the **06_while-loop.py** script as shown above and run it. You should not have any errors and your output should look similar to the output in Step 4b.

Step 5: Create a while loop that checks for a user quit command.

What if we want the program to run as many times as the user wants until the user quits the program? To do this, we can embed the program in a **while** loop that checks if the user enters a quit command, such as **q** or **quit**.

- a. In your **06_while-loop.py** script, make the following changes.
- 1) Add another **while** loop to the beginning of the script which will check for a quit command.
 - 2) Add an if function to the while loop to check for q or quit.

```
while True:
    x=input("Enter a number to count to: (Enter q to quit.)")
    if x == 'q' or x == 'quit':
        break

x=int(x)
y=1
while True:
    print(y)
    y=y+1
    if y>x:
        break
```

- b. Your output should look similar to the following test in which the user entered two different values before quitting the program. The user then restarted the program and tested quitting with **q**.

```
===== RESTART: /home/Student/06_while-loop.py =====
Enter a number to count to: 3
1
2
3
Enter a number to count to: 5
1
2
3
4
```

```
5
Enter a number to count to: quit
>>>
===== RESTART: /home/Student/06_while-loop.py =====
Enter a number to count to: q
>>>
```

Part 6: Explore File Access

In addition to user input, you can access a database, another computer program, or a file to provide input to your program. In Part 6, you will write a program to read an external file. You will then construct a program based on a set of requirements.

Step 1: Create program to access an external file.

The `open()` function can be used to access a file using the following syntax:

```
open(name, [mode])
```

The `name` parameter is the name of the file to be opened. If the file is in a different directory than your script, you will also need to provide path information. For our purposes, we are only interested in three `mode` parameters:

- **r** - read the file (default mode if mode is omitted).
 - **w** - write over the file, replacing the content of the file.
 - **a** - append to the file.
- a. Complete the following steps to read and print a file:
- 1) Create a text file called **devices.txt**. Save the file in the same directory as your Python scripts. On each line of the text file, add a Cisco model. Here is a list you can use:
 - Cisco 819 Router
 - Cisco 881 Router
 - Cisco 888 Router
 - Cisco 1100 Router
 - Cisco 4321 Router
 - Cisco 4331 Router
 - Cisco 4351 Router
 - Cisco 2960 Catalyst Switch
 - Cisco 3850 Catalyst Switch
 - Cisco 7700 Nexus Switch
 - Cisco Meraki MS220-8 Cloud Managed Switch
 - Cisco Meraki MX64W Security Appliance
 - Cisco Meraki MX84 Security Appliance
 - Cisco Meraki MC74 VoIP Phone
 - Cisco 3860 Catalyst Switch
 - 2) Open a blank script and save it as **07_file-access1.py** in the same directory as **devices.txt**.

Lab - Construct a Basic Python Script

- 3) Create a script to read and print the contents of **devices.txt**. After printing the contents of the file, use the **close()** function to remove it from the computer's memory. The **close()** function should not be indented, as shown.

Note: A variable named **file** is created to hold the contents of the file. However, that variable can be called anything the programmer chooses.

```
file=open("devices.txt","r")
for item in file:
    print(item)
file.close()
```

- b. Run the script and troubleshoot, if necessary. Your output should look like the following:

```
===== RESTART: /home/Student/07_file-access1.py =====
Cisco 819 Router

Cisco 881 Router

Cisco 888 Router

Cisco 1100 Router

Cisco 4321 Router

Cisco 4331 Router

Cisco 4351 Router

Cisco 2960 Catalyst Switch

Cisco 3850 Catalyst Switch

Cisco 7700 Nexus Switch

Cisco Meraki MS220-8 Cloud Managed Switch

Cisco Meraki MX64W Security Appliance

Cisco Meraki MX84 Security Appliance

Cisco Meraki MC74 VoIP Phone

Cisco 3860 Catalyst Switch
```

Step 2: Remove the blank lines from the output.

- a. You may have noticed that Python added a blank line after each entry. Remove this blank line using the **strip()** method. This method is available to strings type variables. Edit your **07_file-access1.py** script as shown in the following example:

Lab - Construct a Basic Python Script

```
file=open("devices.txt","r")
for item in file:
    print(item.strip())
file.close()
```

- b. Run your script. The output should not display blank lines between the device names.

Step 3: Copy a file into a list variable.

Most of the time when programmers access an external resource such as a database or file, they want to copy that content into a local variable that can then be referenced and manipulated without impacting the original resource.

- a. Copy the content of the devices.txt file into a Python list using the following steps:

- 1) Create an empty list.
- 2) Use the append parameter to copy file content to the new list.
- 3) Modify your **07_file-access.py** as shown in the following:

```
devices=[]
file=open("devices.txt","r")
for item in file:

    devices.append(item.strip())
file.close()
print(devices)
```

- b. Run your script. Your output should look like the following example:

```
===== RESTART: /home/Student/07_file-access.py =====
['Cisco 819 Router', 'Cisco 881 Router', 'Cisco 888 Router', 'Cisco 1100
Router', 'Cisco 4321 Router', 'Cisco 4331 Router', 'Cisco 4351 Router',
'Cisco 2960 Catalyst Switch', 'Cisco 3850 Catalyst Switch', 'Cisco 7700 Nexus
Switch', 'Cisco Meraki MS220-8 Cloud Managed Switch', 'Cisco Meraki MX64W
Security Appliance', 'Cisco Meraki MX84 Security Appliance', 'Cisco Meraki
MC74 VoIP Phone', 'Cisco 3860 Catalyst Switch']
```

Step 4: CHALLENGE: Create a script to allow a user to add devices to an external file.

What if you want to add more devices to the **devices.txt** file? You can open the file in append mode and then ask the user to provide the name of the new devices.

- a. Complete the following steps to create a script:

- 1) Open a new file and save it as **07_file-access_activity.py**.
- 2) For the **open()** function use the mode **a**, which will allow you to append an item to the **devices.txt** file.
- 3) Inside a **while True:** loop, embed an **input()** function command that asks the user for the new device.
- 4) Set the value of the user's input to a variable named **newItem**.
- 5) Use an **if** statement that breaks the loop if the user types **exit** and prints the statement **"All done!"**.
- 6) Use the command **file.write(newItem + "\n")** to add the new user provided device. The **"\n"** will add a new line to the file for the next entry.

- b. Run and troubleshoot your script until you get output similar to the following example:

```
===== RESTART: /home/Student/07_file-access.py =====
```

Lab - Construct a Basic Python Script

```
Enter device name: Cisco 1941 Router
Enter device name: Cisco 2950 Catalyst Switch
Enter device name: exit
All done!
>>>
```

The following is one way students can implement the requirements of the program.

```
file = open("devices.txt", "a")
while True:
    newItem = input("Enter device name: ")
    if newItem == "exit":
        print("All done!")
        break
    file.write(newItem + "\n")
file.close()
```