Lab - Python Programming Review

***** MODIFIED FOR NETLAB+ *****

Objectives

Part 1: Start Python and VS Code

Part 2: Review Data Types and Variables

Part 3: Review Lists and Dictionaries

Part 4: Review the Input Function

Part 5: Review If, For, and While Functions

Part 6: Review Methods for File Access

Background / Scenario

In this lab, you review basic Python programming skills including data types, variables, lists, dictionaries, user input, if statements, for and while loops, and file access. This lab is not meant as a substitute for prior programming experience and does not necessarily cover all the Python skills you will need for this course. However, this lab should serve as a good measure of your Python programming skills and help direct you to where you may need more review.

Note: This is a reminder. Be sure you observe correct Python indention conventions when writing your scripts. If you need a tutorial, search the internet for “Python indention rules”.

Required Resources

- DEVASC Virtual Machine

Instructions

Part 1: Starting Python and VS Code

In this part, you will review starting Python’s interactive interpreter and using Visual Studio Code to write and run a “Hello World” script.

Step 1: Start Python.

a. To check the Python version running in the VM, open a terminal window and enter the command **python3 -V**. This is a good command to run if you were to install Python on a different computer or needed to check what version is installed.

```
devasc@labvm:~$ python3 -V
Python 3.8.2
```

Note: You would need to change it to **python2 -V** if a different device you are using is running version 2. However, as of January 1, 2020, Python 2 is no longer supported. Therefore, Python 2 is not supported in this lab or this course.

Note: At the time this lab was written, Python 3.8.2 was the latest version. Although you can update your Python install with the **sudo apt-get install python3** command, this lab and the rest of the labs in this course are based on Python 3.8.2.

b. To start Python, type **python3**. The three angle brackets (>>>) indicate that you are in Python's interactive interpreter.

```
devasc@labvm~$ python3
Python 3.8.2 (default, Mar 13 2020, 10:14:16)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Step 2: Use the Interpreter as a calculator.

a. 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 operations.

Operation	Math	Syntax
Addition	a+b	a+b
Subtraction	a-b	a-b
Multiplication	axb	a*b
Division	a÷b	a/b
Exponents	a^b	a**b

Enter a few math operations using the Python syntax, as shown in the examples.

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

b. Recall that Python uses the standard order of operations commonly known as PEMDAS. Mathematical expressions are evaluated in the following order.

Parentheses

Exponents

Multiplication and **D**ivision

Addition and **S**ubtraction

Try entering an expression with a complex order of operations in the interactive interpreter.

Step 3: Use the interactive 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 as long as you enclose the string in either single quotes (') or double quotes (").

a. Type “Hello World!” or ‘Hello World!’ in the interactive interpreter.

```
>>> "Hello World!"
'Hello World!'
>>> 'Hello World!'
'Hello World!'
```

b. The **print** command can be also be used directly in the interactive interpreter.

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

c. To exit the interactive interpreter, enter **quit()**.

```
>>> quit()
devasc@labvm:~$
```

Step 4: Open VS Code and create a script for Hello World.

There are many development environments available for programmers to manage their coding projects. In this course, the labs will use the VM’s installation of Microsoft’s Visual Studio Code (VS Code).

a. Open VS Code. If this is your first time, you will most likely be presented with a **Welcome** window.

b. Feel free to explore the VS Code menus and options on your own time. For now, click **File > New File** to open a new file.

c. In your new file, type the **print** command from the previous step.

d. Save the script as **hello-world.py** in the **labs/devnet-src/python** folder. Be sure you add the extension **.py** for Python file.

e. To run the script, click **Run > Run Without Debugging**. A terminal window opens inside VS Code, runs the code to launch an instance of Python, runs your script, then exits out of Python back to your Linux command line.

```
devasc@labvm:~/labs/devnet-src/python$  env DEBUGPY_LAUNCHER_PORT=36095
/usr/bin/python3 /home/devasc/.vscode/extensions/ms-python.python-
2020.4.76186/pythonFiles/lib/python/debugpy/no_wheels/debugpy/launcher
/home/devasc/labs/devnet-src/python/hello-world.py
Hello World!
devasc@labvm:~/labs/devnet-src/python$
```

f. Now that you have a command line open inside VS Code, you can manually launch Python and quickly run your script with the following command.

```
devasc@labvm:~/labs/devnet-src/python$ python3 hello-world.py
Hello World!
devasc@labvm:~/labs/devnet-src/python$
```

g. You can also open a terminal window outside of VS Code and enter the same command making sure to provide path information.

```
devasc@labvm:~$ python3 ~/labs/devnet-src/python/hello-world.py
Hello World!
devasc@labvm:~$
```

In this course you will typically run your scripts directly inside VS Code.

Part 2: Review Data Types and Variables

In this part, you will use the interactive interpreter to review data types, create variables, concatenate strings, and cast between data types.

Step 1: Use the interactive interpreter to review 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, and so on. 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.

Use the **type()** command to determine the basic data types: int, float, string, Boolean

```
devasc@labvm:~/labs/devnet-src$ python3
Python 3.8.2 (default, Mar 13 2020, 10:14:16)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> type(98)
<class 'int'>
>>> type(98.6)
<class 'float'>
>>> type("Hi!")
<class 'str'>
>>> type(True)
<class 'bool'>
```

Step 2: Review different 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 interpreter, try out the different Boolean operators.

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

Step 3: Use the interpreter to 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 value. For example, create and use the following variable in the interactive interpreter.

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

Step 4: Use the interpreter to 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**".

a. Enter the following four variables and then concatenate them 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
```

b. To print the variables without using a variable to create the space, separate the variables with a comma.

```
>>> print(str1,str2,str3)
Cisco Networking Academy
```

Step 5: Reviewing casting and printing different data types.

a. Converting between data types is called type casting. Type casting often needs to be done in order to work with different data types. For example, concatenation does not work when joining different data types.

```
>>> x=3
>>> print("The value of x is " + x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>>
```

b. Use the **str()** function to convert the integer data type to a string data type.

```
>>> 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. To convert the data type, reassign the variable to the new data type.

```
>>> 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, you can use f-strings and the **"{:.2f}".format** function.

Note: Search the internet to learn more about f-strings and the format function.

```
>>> num=22/7
>>> f"The value of num is {num}"
'The value of num is 3.142857142857143'
>>> pi="{:.2f}".format(num)
>>> f"The value of pi is {pi}."
'The value of pi is 3.14.'
>>>
```

Part 3: Review Lists and Dictionaries

In this part, you will review the methods for creating and manipulating lists and dictionaries.

Step 1: Create and manipulate a list.

a. In programming, a list variable is used to store multiple pieces of ordered information. Lists are also called arrays in some programming environments.

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

The following example shows how to create a list variable called **hostnames**.

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

b. An item in a list can be referenced and manipulated using its index.
   - The first item in a list is indexed as zero, the second is indexed as one, and so on.
   - The last item can be referenced with index **[-1]**.
   - Replace an item by assigning a new value to the index.
   - 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 and manipulate a dictionary.

a. Dictionaries are unordered lists of objects. Each object contains a key/value pair.
   - Create a dictionary using the braces { }.
   - Each dictionary entry includes a key and a value.
   - Separate a key and its value with a colon.
   - Use quotes for keys and values that are strings.

   Create the following dictionary called **ipAddress** with three key/value pairs to specify the IP address values for three routers.

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

b. Unlike lists, objectives inside a dictionary cannot be referenced by their sequence number. Instead, you reference a dictionary object using its key.
   - The key is enclosed with brackets [ ].
   - Keys that are strings can be referenced using single or double quotes.
   - Use a **key** in the **dictionary** statement to verify if a key exists in the dictionary.
   - Add a key/value pair by setting the new key equal to a value.

```
>>> 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'}
>>>
```

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.

```
>>> 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']}
>>>
```

Part 4: Review the Input Function

In this part, you will review how to use the input function to store and display user-supplied data.

Step 1: Create a variable to store user input and then display the value.

Most programs require some type of input either from a database, another computer, mouse clicks, or keyboard input. For keyboard input, use the **input()** function 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. Assign the **input()** function to a variable that asks the user for input and then print the value of the user's input.

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

Step 2: Create a script to collect personal information.

Create and run a script to collect personal information.

a. Open a blank script file and save it as **personal-info.py** in the **~/labs/devnet-src/python** folder.

b. Create a script that asks for four pieces of information such as: first name, last name, location, and age.

c. Add a print statement that combines all the information in one sentence.

d. Your script should run without any errors, as shown in the following output.

```
devasc@labvm:~/labs/devnet-src$ python3 person-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.
devasc@labvm:~/labs/devnet-src$ ^C
```

Part 5: Review If, For, and While Functions

In this part, you review how to create if statements as well as for and while loops.

Step 1: Create an if/else function.

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 are carried out.

a. Open a blank script and save it as **if-vlan.py**. Type the following script into the file.

```
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.")
```

Note: In Python, use four spaces to indent. If you save your file in VS Code, this four space indention will be automatic. When beginning the else statement, make sure to backspace to the left margin.

b. Save the script and run it. Your output should look like the following example.

```
The native VLAN and the data VLAN are different.
```

c. 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.

```
The native VLAN and the data VLAN are the same.
```

Step 2: Create an if/elif/else function.

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.

a. Create this script for your files. Open a blank script and save it as **if-acl.py**. Copy the script into the file.

```
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.")
```

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.

b. Run multiple times to test each statement.

```
devasc@labvm:~/labs/devnet-src/python$ python3 if-acl.py
What is the IPv4 ACL number? 10
This is a standard IPv4 ACL.
devasc@labvm:~/labs/devnet-src/python$ python3 if-acl.py
What is the IPv4 ACL number? 110
This is an extended IPv4 ACL.
devasc@labvm:~/labs/devnet-src/python$ python3 if-acl.py
What is the IPv4 ACL number? 200
This is not a standard or extended IPv4 ACL.
devasc@labvm:~/labs/devnet-src/python$
```

Step 3: Create a for loop.

The Python **for** function 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 in the interactive interpreter to see how a for loop works. The variable name **item** is arbitrary and can be anything the programmer chooses. Often, programmers will shorten this to just the letter **i**.

Note: Be sure you enter four spaces to indent the **print()** function and press the Enter key twice to exit the **for** loop.

```
devasc@labvm:~/labs/devnet-src/python$ python3
Python 3.8.2 (default, Mar 13 2020, 10:14:16)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.>>>
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. Continuing with the same Python instance, enter the following in the interactive interpreter.

Note: Be sure you enter four spaces to indent the **if** function and the eight spaces to indent the **print()** function. Press the Enter key twice to exit and execute the for loop.

```
>>> 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. Enter the following example to see how to use the **append()** method to create a new list called switches. Be sure to follow the indention requirements.

```
>>> switches=[]
>>> for item in devices:
...     if "S" in item:
...         switches.append(item)
...
>>> switches
['S1', 'S2']
>>>
```

Step 4: 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.

a. Open a blank script and save it as **while-loop.py**. Use a while loop to create the following program that counts from a programmed value up to a user-supplied number. The program does the following:

- Asks the user for a number.
- Casts inputted string value to an integer: **x = int(x)**.
- Sets a variable to start the count: **y = 1**.
- While **y <= x**, prints 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. Save and run your script. You should get output like the following.

```
devasc@labvm:~/labs/devnet-src/python$ python3 while-loop.py
Enter a number to count to: 10
1
2
3
4
5
6
7
8
9
10
devasc@labvm:~/labs/devnet-src/python$
```

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. Modify the **while-loop.py** script as shown in the following:

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

d. Save and run your script. You should get the same output as in Step 4b above.

Step 5: Use a while loop to check 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. Modify your **while-loop.py** script with the following changes:

- Add another while loop to the beginning of the script which will check for a quit command.
- Add an if function to the while loop to check for **q** or **quit**.

Note: Be sure you include all the required levels of indention.

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

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

b. Save and run your script. Your output should look similar to the following in which the user entered two different values before quitting the program.

```
devasc@labvm:~/labs/devnet-src/python$ python3 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
devasc@labvm:~/labs/devnet-src/python$
```

Part 6: Review Methods for File Access

In this part, you review methods for accessing, reading, and manipulating a file.

Step 1: Create a program that reads an external file.

In addition to user input, you can access a database, another computer program, or a file to provide input to your program. 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. Open a blank script and save it as **file-access.py**.

b. Create the following program to read and print the content of the **devices.txt** which is in the same directory as your program. After printing the contents of the file, use the **close()** function to remove it from the computer's memory.

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

Note: The contents of the file are set to a variable named file. However, that variable can be called anything the programmer chooses.

c. Save and run your program. You should get output similar to the following.

```
devasc@labvm:~/labs/devnet-src/python$ python3 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
devasc@labvm:~/labs/devnet-src/python$
```

Step 2: Remove the blank lines from the output.

You may have noticed that Python added a blank line after each entry. We can remove this blank line using the **strip()** method.

a. Edit your **file-access.py** program to include the **strip()** method.

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

b. Save and run your program. You should get output similar to the following.

```
devasc@labvm:~/labs/devnet-src/python$ python3 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
devasc@labvm:~/labs/devnet-src/python$
```

Step 3: Copy the content of a file into a list variable.

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

The **devices.txt** file is a list of Cisco devices that can easily be copied into a Python list using the following steps:

- Create an empty list.
- Use the **append** parameter to copy the file content to the new list.
- Print the list.

a. Modify your file-access.py as shown in the following

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

b. Save and run your program. You should get output similar to the following.

```
devasc@labvm:~/labs/devnet-src/python$ python3 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']
devasc@labvm:~/labs/devnet-src/python$
```

Step 4: Challenge: Create a script to allow the user to add devices.

What if you want to add more devices to the **devices.txt** file? You can create a program to open the file in append mode and then ask the user to provide the name of the new devices. Complete the following steps to create a script:

1) Open a new file and save it as **file-access-input.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.
7) Close the file to release it from computer memory.

Run and troubleshoot your script until you get output similar to the following.

```
devasc@labvm:~/labs/devnet-src/python$ python3 file-access-input.py
Enter device name: Cisco 1941 Router
Enter device name: Cisco 2950 Catalyst Switch
Enter device name: exit
All done!
devasc@labvm:~/labs/devnet-src/python$
```

Verify the script by running your program **file-access.py** again and you get output similar to the following.

```
devasc@labvm:~/labs/devnet-src/python$ python3 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',
'Cisco 1941 Router', 'Cisco 2950 Catalyst Switch']
devasc@labvm:~/labs/devnet-src/python$
```