In programming, literals are used to represent constant values. Different types of literals are used to represent different types of values. Variables are names that reference values that may change over time. The three primary classes of literals in Python are boolean literals, numbers, and strings. Boolean literals can only have the values true or false.

**Numbers**

In general there are two types of numbers with Python:

* **Integers**. Integers are whole numbers, which can be very large, and which can also be negative. You can also use other numerical bases with integers, such as binary, octal, and hexadecimal.

The following examples show 27 represented in other bases.

* + **Binary:** 0b00011011
  + **Octal:** 0o33
  + **Hexadecimal:** 0x1B
* **Floats**. Python provides the floating-point data type for precise numerical values such as 1.23. Floats can also use exponential notation. For example, 1.0e6 which is 1,000,000.

**Strings**

Strings are a series of characters. The characters in a string can be alphanumeric including many special characters such as punctuation. A few important points about strings:

* **Quotes**. Strings can be denoted by either single (') or double (") quotation marks. For example, **'this is my string'**. However, you must be consistent. If you need to have one type of quote as actual data within a string, you can use the other type of quote to delimit the string. If you wish to extend a single string across multiple lines, you can use triple quotes at the beginning and ending of the string.
* """This is an example of a string that spans more than one line.
* Denoting the string with three quotation marks allows your strings
* to span multiple lines.

"""

* **Escape characters.** If you need to represent something that might cause problems within a string, such as quotes or newline or carriage return, you can use a backslash ('\') character, which is called 'escaping' the character. Two examples are \n and \r, which represent newline and carriage return.
* >>> my\_string = "This is a test of a newline. \nIt inserts a line when the string is printed."
* >>> print my\_string
* This is a test of a newline.
* It inserts a line when the string is printed

>>>

* **Concatenation.** Strings can be concatenated using the addition symbol '+'. For example, 'my' + 'string'. You may need to add a space between the strings to be concatenated, but when you print, a space is added for you automatically.
* **Single character.** You can get a single character from a string by specifying its offset, or index, in square brackets 'my\_string[n]'. For example, if my\_string = 'Cisco', then my\_string[2] would be the character 's'. Remember that in Python, indexes start at 0.
* **Slicing.** You can get a specific slice or subset of a string using offsets inside square brackets. With slices, you specify the start index, the end index, and the step. For example, my\_string[0::2] would result in every second character from my\_string. Default for the start index is 0, default for the end index is the last character in the string, and default for the step is 1.
* >>> my\_string = "Cisco Systems"
* >>> print my\_string[0::2]
* CsoSses
* >>> print my\_string[0:5:1]
* Cisco
* >>> print my\_string[6::]

Systems

* **Split.** You can split a string into a list of strings using the 'split' method, which will split the string using the split character you provide. For example, for a string with a [CSV](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CSV), you can type my\_string.split(',') to split the string into a list of the separated values.
* >>> my\_string = "router, switch, access point"
* >>> my\_string.split(',')

['router', ' switch', ' access point']

You can also combine string operations. For example, you can combine the split method and slicing:

>>> device1 = my\_string.split(',')[0]

>>> print device1

router

* **Strip.** You can use strip() to remove white space from the beginning or end of a string. In the split example, switch and access point both have a leading space. The strip method can be used to strip these leading spaces.
* >>> my\_string = "router, switch, access point"
* >>> my\_string.split(',')
* ['router', ' switch', ' access point']
* >>> device2 = my\_string.split(',')[1]
* >>> print device2
* switch
* >>> device2 = device2.strip()
* >>> print device2

switch

* **Length.** You can get the length of a string using len().
* >>> my\_string = "Cisco Systems"
* >>> len(my\_string)
* 13

>>>

There are several other string operations including join(), replace(), duplicate(), find(), and many others. Consult the Python documentation and other sources for a complete list and examples.

**Variables, Objects, and References**

Consider your 'hello device' application.

ping = pexpect.spawn('ping –c 5 localhost')

The statement is creating a variable that is called 'ping', which is a reference to the object created by the 'pexpect.spawn(...)' method call. The result is that in 'ping' there is a reference to this object, which you can then use to call the 'expect' method.

result = ping.expect([pexpect.EOF, pexpect.TIMEOUT])

Note also that 'result' is also a variable pointing at the object returned by the 'ping.expect()' call.

It is important to understand that in Python everything is an object and variables are references to objects. When you create a variable in Python, you are creating a reference to the actual object you have created. So when you say:

>>> a=5

You are creating a variable that is named 'a', which points to the object which contains the value '5'. If you then declare another variable 'b':

>>> b=a

You are creating another variable and having it reference the same object that 'a' references.

>>> a

5

>>> b

5

If you later change the value that 'a' points to, Python creates another object, and now the values of 'a' and 'b' are referencing different objects.

>>> a=a+2

>>> a

7

>>> b

5

**Printing**

During development and with a finished application, it will be important to print results of your program. There are two methods for printing variables:

* **Print statement.** Using the 'print' function is suitable either during development to print state information, or as the output results of your application. Printing can be constant strings or variables which can be substituted or formatted to your satisfaction. One thing to note is that 'print' changed between Python 2 and 3. With Python 2, print was a keyword which could be invoked without parenthesis '()'. With Python 3, print became a function that you call, thus requiring parenthesis and the passing of parameters. Since the lab environment is using Python 2.7, you can use either method for printing.
* **Automatic.** With Python, it is also possible to have variables or constants be printed automatically, just by listing them as a statement. Note that the output includes quotes for strings:
* >>>my\_string = "Cisco Systems"
* >>>my\_string
* 'Cisco Systems'

>>>

[Terms & Conditions](http://www.cisco.com/web/siteassets/legal/terms_condition.html) [Privacy Statement](https://www.cisco.com/c/en/us/about/legal/privacy-full.html) [Cookie Policy](https://www.cisco.com/c/en/us/about/legal/privacy-full.html#cookies) [EULA](https://learningnetworkstore.cisco.com/EULA.html) [Trademarks](http://www.cisco.com/web/siteassets/legal/trademark.html) [Keyboard Shortcuts](https://ondemandelearning.cisco.com/cisco-cte/prne/sections/4/pages/3)

Programming languages have different ways of defining code blocks, which are segments of code that get executed as a result of loop iterations, conditional statements, and functions. In contrast to some other languages, in Python code blocks do not have 'begin' and 'end' delimiters such as curly brackets ('{}'). Instead, Python has two components that combine to define a code block:

* **Colon (':').** The statement preceding the block will end with a colon (':').

if len(my\_string) > 10:

* **Indentation.** Python uses indentation to identify statements in a code block. Indentation must be uniform and consistent; and when the indentation ends, the code block ends.
* if len(my\_string) > 10:

print my\_string

This indentation scheme may seem strange at first, but consider that other more flexible schemes for identifying code blocks are well-suited to computers, but less suited to humans. You can easily create mis-matched curly brackets which are syntactically correct, but semantically flawed, due to misplaced brackets. This is not an issue for Python.

**Note**

In Python, it is highly suggested you use spaces for indentation. While tabs will work, they can cause readability problems between different platforms and editors. The key thing to remember is being consistent about your indent choice so that your program is more readable.

## Importance of Comments

Programmers have differing opinions regarding comments: their usefulness, their reliability, their value to novice and experienced developers. Some would say that the code itself should be self-documenting; and some would say that comments should constitute most of non-whitespace characters in an application. Most programmers are somewhere in between.

But there are a few aspects of commenting code that most programmers can agree on:

* **Explaining:** Comments should be used to explain the purpose of a line or section of code. Comments such as **x = y # set x equal to y** are not very useful.

routes = {} # create dictionary to hold routing info

Has more value to the person hoping to understand the code.

* **Maintenance:** Comments should be maintained, meaning that if the code changes, make sure the comments related to the code change as well. An over-abundance of unnecessary comments makes maintenance overly difficult.
* **Auto-generation:** Most programming languages have some type of automatic documentation generation. Java has Javadocs, and Python has Pydoc. Create your comments in such a way as to facilitate this capability.

## Comments

In Python, comments can come in two forms: single-line and multi-line. In brief:

### Single-Line Comments

* Denoted by '#'; everything following, up to the end of the line, is considered part of the comment.
* Useful for quick explanation of the purpose of a piece of code if it is not obvious.
* Useful for describing a step-by-step process, inline with the code, to facilitate understanding of the overall sequence of what is going on.
* Comments stating the obvious are of little value, and actually may be detrimental, because of the overhead of maintaining such things.

### Multi-line Comments

* Use triple-quote marks to identify the beginning and end of a multi-line comment.
* Useful for module and function descriptions at the beginning.
* For modules, include the purpose of the module, and what it contains.
* For functions, include the purpose of the function, and also the parameters that are passed, and the values returned, by that function.
* Multi-line comments will be gathered by the 'pydoc' tool when run against your application or code, and will generate a nicely formatted, easy-to-read piece of documentation regarding your module or function.

But be aware that that documentation will only be as good as what you put there. Meaningless multi-line comments will result in nicely formatted, meaningless documentation.

Here is an example of well-documented code , with single and multi-line comments:

#!/usr/bin/env python

"""

Example of a script that executes a CLI command on a remote

device over established SSH connection.

Administrator login options and CLI commands are device specific,

thus this script needs to be adapted to a concrete device specifics.

Current script assumes interaction with Cisco IOS device.

NOTES: Requires installation of the 'paramiko' Python package:

pip install paramiko

The 'paramiko' package is documented at:

http://docs.paramiko.org

Complete set of SSH client operations is available at:

http://docs.paramiko.org/en/1.15/api/client.html

command\_ssh.py

"""

# built-in modules

import time

import socket

# third-party modules

import paramiko

def enable\_privileged\_commands(device\_info, rsh):

"""Turn on privileged commands execution.

:param dict device\_info: dictionary containing information

about target device.

:param paramiko.channel.Channel rsh: channel connected to a remote shell.

"""

cmd = "enable\n"

# Execute the command (wait for command to complete)

rsh.send(cmd)

time.sleep(1)

output = rsh.recv(device\_info['max\_bytes'])

if(device\_info['password\_prompt'] in output):

password = "%s\n" % device\_info['password']

rsh.send(password)

rsh.recv(device\_info['max\_bytes'])

Pydoc is a documentation tool for Python and is included in Python versions 2.1 and higher. Pydoc uses multiline comments in modules and functions. The pydoc output is generated by the following commands:

$ **pydoc command\_ssh.py**

Or

$ **pydoc command\_ssh**

**Note**

Some versions of Python and pydoc require the file suffix (.py), and some do not. The best advice is to try one method, and if it fails, try the other.

Running pydoc against well-documented code will result in documentation such as shown below:

Help on module command\_ssh:

NAME

command\_ssh

FILE

/home/PRNE/Module6/command\_ssh.py

DESCRIPTION

Example of a script that executes a CLI command on a remote

device over established SSH connection.

Administrator login options and CLI commands are device specific,

thus this script needs to be adapted to a concrete device specifics.

Current script assumes interaction with Cisco IOS device.

NOTES: Requires installation of the 'paramiko' Python package

pip install paramiko

The 'paramiko' package is documented at:

http://docs.paramiko.org

Complete set of SSH client operations is available at:

http://docs.paramiko.org/en/1.15/api/client.html

FUNCTIONS

check\_config\_mode(device\_info, rsh)

Check if CLI on the device is in configuration mode.

:param dict device\_info: dictionary containing information

about target device.

:param paramiko.channel.Channel rsh: channel connected to a remote shell.

:return: True if CLI is in configuration mode, False otherwise

Returns a boolean

connect\_ssh(device\_info)

Establish SSH connection to a remote device.

:param dict device\_info: dictionary containing information

about target device.

:return: an instance of paramiko.SSHClient class connected

to the device on success, None on failure.

disable\_paging(device\_info, rsh)

Disable CLI paging on a remote device.

:param dict device\_info: dictionary containing information

about target device.

:param paramiko.channel.Channel rsh: channel connected to a remote shell.

## Learning Lab Procedure

**Step 1**

Open a text editor and enter the same statements used in the previous procedure.

|  |
| --- |
| import pexpect  ping = pexpect.spawn('ping -c 5 localhost')  result = ping.expect([pexpect.EOF, pexpect.TIMEOUT])  print (ping.before) |

**Step 2**

Save your file with the name hello-device.py and exit the text editor.

**Step 3**

At the command line, run your Python program by typing 'python hello-device.py'.

|  |
| --- |
| $ python hello-device.py |

After a few moments, the output from your application will be printed to your terminal window.

|  |
| --- |
| >>> print(ping.before)  PING localhost (127.0.0.1): 56 data bytes  64 bytes from 127.0.0.1: icmp\_seq=0 ttl=64 time=0.054 ms  64 bytes from 127.0.0.1: icmp\_seq=1 ttl=64 time=0.054 ms  64 bytes from 127.0.0.1: icmp\_seq=2 ttl=64 time=0.087 ms  64 bytes from 127.0.0.1: icmp\_seq=3 ttl=64 time=0.114 ms  64 bytes from 127.0.0.1: icmp\_seq=4 ttl=64 time=0.122 ms  ... |

Chapter 5 Reading and Writing Network Device Information

5.1 Introduction

This lesson covers simple input and output of text files, which is useful for such tasks as:

* Reading device information (IP address, username/password, interface information, router information, and so on) from a file, in order to perform operations on a group of routers and/or switches.
* Reading device commands that are of interest to the program.
* Writing the results of such operations.

Therefore, this lesson will focus on the following topics:

* Basics of reading and writing files
* Reading input from a file
* Reading input from user input
* Writing output to a file

5.2 Input/Output Basics

Dealing with normal text files will involve the following actions by your application:

* **Opening the file.** The file will need to be opened before you can read or write to it. When opening a file, you will specify the filename, and your intended mode—read, write, or both.
* **Closing the file.** When your application has completed its file operations, be sure to close the file. Leaving the file open may not cause problems for a while, but lingering references to files that have not been appropriately closed can cause problems with your operating system over time.
* **Filename.** When opening a file, you will provide the filename. You will need to specify the path to the file as well; it is a good idea to consider using relative paths, rather than fixed ones, so that your application can run wherever it is installed.
* **Mode.** The intended operations to be performed on the file must be specified, such as read, write, or append. Read is the default operation. You will also need to specify if the file is binary. The default file type is text.

Reading from a file is simple and getting the actual contents can be done in multiple ways:

* **Open.** The file must first be opened.

my\_input\_file = open("/home/Documents/input.txt", 'r')

* **Read.** There are multiple options for reading the file:
  + **readlines()**. Allows you to read all lines in the file into your read buffer.

my\_input = my\_input\_file.readlines()

* + **readline()**. Allows you to read one line at a time into your read buffer.

my\_input = my\_input\_file.readline()

* + **read()**. Allows you to read a specific number of characters into your read buffer.

my\_input = my\_input\_file.read(10)

If you choose to not bother with specific read operations, and want to iterate over the entire file one text line at a time, you can use a basic **for** loop. In exploring Python, you will learn about control structures, such as **for** loops. For now, you will need to know how to read through a file, line by line, using a **for** loop. The code to read a file is:

my\_file = open('myfile', 'rt')

for line in my\_file:

# code block to handle text data in 'line'

This code opens the file, then immediately jumps into a **for** loop, which will iterate over all the lines in the file, one at a time. At each iteration, the line of the text file is placed into the variable that is called line, to be used in whatever functionality the code block is implementing.

Reading input that comes from a user at a terminal running the application is simpler than reading from a file. There is no need to do an 'open' or to iterate over lines of text. In the simplest case, use the built-in 'input' function. Provide the text to print to the user to let them know what to enter, and use a variable to receive the reference to the object containing whatever the user entered.

In the example, the user will see the message **Input device name:**. Once they have done so and pressed 'enter', the value they input will be pointed to by the variable 'my\_input'.

>>> my\_input = input('Input device name: ')

Input device name: "Branch1"

>>>print my\_input

Branch1

>>>

Writing to a file is much like reading:

* **Open.** Open the file for writing or appending.

my\_output\_file = open("/home/documents/text.txt", 'w')

**Note**

Opening a file with the write argument will create an empty file. If you are opening an existing file, use the append argument if you want to keep the existing file contents.

|  |
| --- |
| my\_output\_file = open("/home/documents/text.txt", 'a') |

* **Write.** There are also options for writing to a file:
  + **write(data)**. To write all data at once.
  + my\_string = "Cisco Systems"
  + my\_output\_file = open("/home/documents/text.txt", 'w')
  + my\_output\_file.write(my\_string)

my\_output\_file.close()

* + **writelines(data)**. To write a few lines of data.
  + my\_string = "This is line 1"
  + my\_output\_file = open("/home/documents/text.txt", 'w')
  + my\_output\_file.writelines(my\_string)

my\_output\_file.close()

It is also possible to write things out to a file using the 'print' function:

print(data, filename)

Chapter 6: Communicating with Network Devices

At this point in the course, it is time to examine communication with networking devices. The three main methods for communication are:

Direct to a device using CLI commands:

* Using CLIs is interesting because they are ubiquitous (everywhere).
* You must create applications specific to each different device type, including differences across software revisions.
* The data is unstructured, making it difficult to input correctly, or to parse output.
* With CLI, you only operate on one device at a time.

Direct to a device using an API:

* Device-level APIs are not as ubiquitous as CLI, but are becoming more prevalent.
* You must create applications specific to the API.
* The data is structured, making it easier to work with than CLI.
* With device-level APIs, you only operate on one device at a time.

Via SDN Controller:

* Limitation – not every device supports communication via an SDN controller.
* APIs tend to be more common, although certain ones, such as NETCONF are still device-specific.
* Generally, the data is structured for inputting configuration data, and for receiving operational data.
* Communication via an SDN controller opens up the possibility of dealing with multiple devices at the same time.

Note that this course is primarily about Python and not about the specifics of communication via one mechanism or the other. Therefore, the labs will use either Telnet or SSH to interact with network devices.

If communication via [CLI](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CLI)had to be coded in your application, it would be quite difficult. Fortunately, in keeping with Python's philosophy of providing a rich set of libraries for various tasks, there are two specific tools, Pexpect and Paramiko, that will assist you in device communication.

Pexpect:

* Pexpect is a general tool for spawning processes, hence you can do many things such as 'ping', 'telnet', and 'SSH'. Specifically, Pexpect is used for connecting to devices and interacting with their sometimes complicated user interfaces.
* The login process must be done manually.
* Pexpect can be a little verbose, but that is the general manner of the tool – it can be verbose but it allows the application to anticipate ('expect') certain replies, and respond accordingly.

Paramiko:

* Paramiko is a Python implementation of [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH) supporting both client and server functionality.
* Supports automated login process.
* Manual checking of results using pattern matching to determine results.

Communicating with a device via[CLI](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CLI) requires a telnet or [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH) connection. Using Pexpect, establishing a telnet connection requires the following process:

## Import Pexpect

Pexpect is not one of the built-in functions for Python, so you will have to 'import' it into your program.

import pexpect

The import command tells Python that your code may be using the functions, objects, and methods that come with the Pexpect library. Your code will be connecting to a network device at a specific [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address), using a specific username and password. In the Pexpect examples that follow, this information will be stored in variables:

ip\_address = '10.30.30.1'

username = 'cisco'

password = 'cisco'

## Connect Via Telnet Session

You will create ('spawn') a Pexpect session using the **pexpect.spawn(...)** method:

# Create Pexpect telnet session

session = pexpect.spawn('telnet ' + ip\_address, timeout=20)

The **spawn** method creates a new session, using the command given inside the quotation marks. In this case, the command is 'telnet', passing in the IP address of the destination network device. The **expect** method tells the program what to expect as a result of the connection request. The following code shows how to 'expect' a response to the connection request and store the response in a variable named 'result':

result = session.expect(['Username:', pexpect.TIMEOUT])

# Check for error, if so then print error and exit

if result != 0:

print '--- FAILURE! creating session for: ', ip\_address

exit()

In the example, the session object expects the device to respond with either 'Username:' or a timeout. If the result is not equal to 0 – the first item in our list of possible responses – then that indicates an error, and the code prints an error message and exits. If the result is 0, the telnet connection was successful and the process continues by entering the username and password. Your application must send the login credentials for the device which is done using the **sendline** command, passing a string containing the value of your desired command. For the login process, the code would look as follows:

# Session expecting username, enter it here

session.sendline(username)

result = session.expect(['Password:', pexpect.TIMEOUT])

# Check for error, if so then print error and exit

if result != 0:

print '--- FAILURE! entering username: ', username

exit()

# Session expecting password, enter it here

session.sendline(password)

result = session.expect(['>', pexpect.TIMEOUT])

# Check for error, if so then print error and exit

if result != 0:

print ' FAILURE! entering password: ', password

exit()

print '--- Success! connecting to: ', ip\_address

print '--- Username: ', username

print '--- Password: ', password

print '------------------------------------------------------\n'

In the example, the application sends the username, expecting to receive the 'Password' prompt, then sends the password, expecting the CLI prompt ('>') for an [IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS) device. If everything works successfully, a message to that effect is printed.

## Sending CLI Requests

The general model of Pexpect interaction is as follows:

* Issue a command
* Tell Pexpect to 'expect' a list of potential responses
* Take action based on the actual response

For example:

# Send 'show interface' command to gather data about links

session.sendline('show interface summary') # Send 'show interface' command

result = session.expect(['>', pexpect.TIMEOUT, pexpect.EOF])

if result == 0: # successful reply to show interface command

print 'Command sent successfully'

return True

elif result == 1: # timeout occurred

print 'Timed out'

return False

elif result == 2: # EOF occurred

print 'Received unexpected EOF response'

return False

else:

print 'Unexpected response'

return False

The **elif** statement is used to select between multiple options, similar to the **case** statement in other languages.

## Receiving CLI Responses

Once your CLI request has completed successfully, your application will often receive output data. Output data is provided to your application in the 'before' member variable of your session:

show\_int\_output = session.before

The output is received in the form of a string. If the output consists of multiple lines, your application may use the **splitlines()**function to separate the string into a list of strings, one item for each line of output.

int\_output\_lines = show\_int\_output.splitlines()

If the output is more complex, it may require parsing using something such as regular expressions.

## Close Connection

When you are done interacting with your device, you should send the appropriate CLI commands to exit the Telnet connection on the device. You should also close the session within the application to free up local resources..

session.close()

Connecting to a device using [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH) using Pexpect is very similar to connecting via Telnet. The major difference is in the command that is used when you launch the session.

## Connect Via SSH Using Pexpect

To create ('spawn') a Pexpect session using the pexpect.spawn(...) method:

# Create pexect SSH session

session = pexpect.spawn('ssh ' + username + '@' + ip\_address, timeout=20)

The 'spawn' method creates a new session, just as with telnet, giving you a session for sending commands, receiving and parsing output, and so on. The rest of the login process is the same as for telnet, with the exception being that there is no need to have code to enter the 'username', since that has already been provided in the SSH command line itself.

## Connect Via SSH Using Paramiko

Paramiko is a Python implementation of [SSHv2](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSHv2) that provides both client and server functionality. The following is an example of establishing an SSH connection to a device using Paramiko:

import paramiko

ip\_address = '10.30.30.1'

username = 'cisco'

password = 'cisco'

print '\n------------------------------------------------------'

print '--- Attempting paramiko connection to: ', ip\_address

# Create paramiko session

ssh\_client = paramiko.SSHClient()

# Must set missing host key policy since we don't have the SSH key

# stored in the 'known\_hosts' file

ssh\_client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

# Make the connection to our host.

ssh\_client.connect(hostname=ip\_address,

username=username,

password=password)

# If there is an issue, paramiko will throw an exception,

# so the SSH request must have succeeded.

print '--- Success! connecting to: ', ip\_address

print '--- Username: ', username

print '--- Password: ', password

print '------------------------------------------------------\n'

Once the SSH connection is established, commands can be executed using the **ssh\_client.exec\_command** function which uses three variables: **stdin**, **stdout**, and **stderr**. **stdin** passes the command to the device, **stdout** is used to store the response, and **stderr** is used to report any errors. For example, to obtain the [IP](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP)route table from a router, you would enter:

stdin, stdout, stderr = ssh\_client.exec\_command('show ip route')

ip\_route\_table = stdout.readlines()

ssh\_client.close()

To close the SSH session use **ssh\_client.close()**

Data Structure

There are 3 common steps with most scripts you create:

* Read in a set of information, such as a list of MAC addresses in a switch forwarding table.
* Analyze the information, in some way such as calculating the number of MAC addresses that are reachable via each interface.
* Output the information, such as printing a list of interfaces and the count of MAC address to the screen for the user to see.

In between these steps, you will need to store information temporarily until you are ready to output the results.

For example, to build a script that counts the MAC addresses associated with each interface, you need to read in the MAC address table from a switch:

Vlan Mac Address Type Ports

---- ----------- -------- -----

11 fa16.3e14.a0f5 DYNAMIC Gi0/2

11 fe00.6abe.ff01 DYNAMIC Gi0/2

14 0050.56be.2548 DYNAMIC Gi0/1

14 0410.50f2.3f44 DYNAMIC Gi0/1

14 0072.7f33.debe DYNAMIC Gi0/1

As your script reads in the lines of data, it will store the interface names that are seen in the MAC address table and a count of the number of MAC addresses associated with each interface. The information that it stores would look like similar to:

Interface Count of MAC Addresses

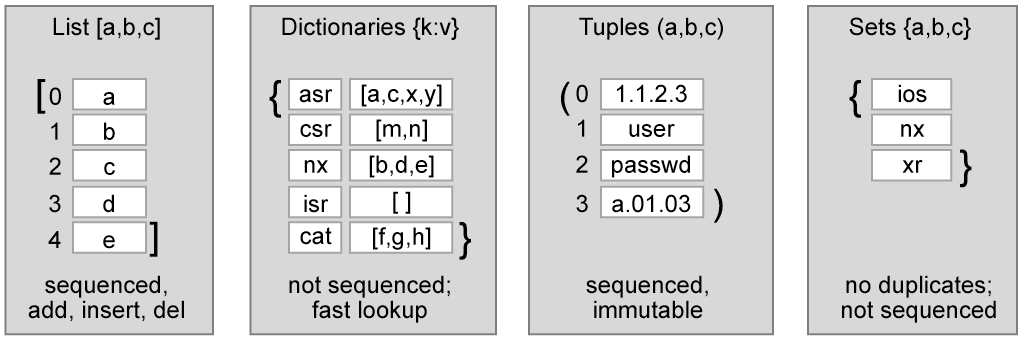
Gi0/1 3

Gi0/2 2

Each time your script reads a line from the MAC address table, it will increment the counter associated with the interface name. This information needs to be stored until the script has finished processing all entries in the MAC address table and you are ready to output the final totals to the user.

In programming, the mechanism for storing data temporarily inside your program is called a data structure. There are different kinds of data structures that are useful for storing different kinds of data and accessing the data in different ways. Python supports a rich set of data structures that are useful for storing various types of information. The data structures are:

* **Lists:**sequences of items, which are indexed by number. Items can be heterogeneous (of different types). Items can be added to a list, removed from a list, and inserted into a list.
* **Dictionaries:** unordered tables of key-value pairs, which are indexed by the value of the key. Items can be added and removed from a dictionary.
* **Tuples:** sequences of items like lists, but a tuple is immutable, meaning its items cannot be changed.
* **Sets:** unordered collections of items. Items can be added or removed from a set, but there can be no duplicate values.



A list is a simple data structure that stores a set of items in a list that is ordered by sequence number (0, 1, 2, 3, 4). If you wanted to simply store the list of all MAC addresses in the table, you could store this information in a list:

[

0 'fa16.3e14.a0f5',

1 'fe00.6abe.ff01',

2 '0050.56be.2548',

3 '0410.50f2.3f44',

4 '0072.7f33.debe'

]

A dictionary is a bit like a list, but instead of identifying the items in the list using a sequence number (such as item #2 is 0050.56be.2548), the items in a dictionary are identified by their name. The item name is also called a key.

You can store the list of interface names seen in the MAC address table in a dictionary. The key in this dictionary would be the interface name and the value would be the number of MAC addresses associated with the interface:

{

'Gi0/1':'3',

'Gi0/2':'2'

}

Each time your script encounters an entry in the MAC address table for interface Gi0/1, it can look up the entry in the dictionary by name (such as Gi0/1) and increment the value by 1. A dictionary is useful in this scenario because you can reference the entry in the dictionary by name.

Data structures can also be combined to provide more flexibility. If you are writing a script that lists the MAC addresses associated with each interface in addition to keeping a count, you could use a dictionary where the key is the interface name and the value is the list of MAC addresses associated with the interface.

{

'Gi0/1':[

0 '0050.56be.2548',

1 '0410.50f2.3f44',

2 '0072.7f33.debe'

],

'Gi0/2':[

0 'fa16.3e14.a0f5',

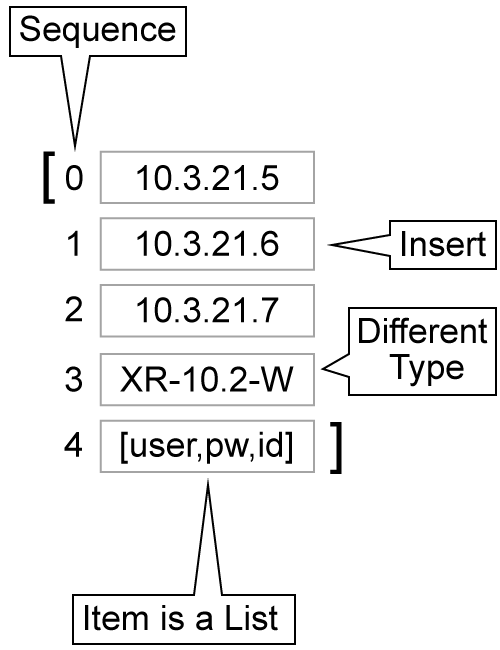
1 'fe00.6abe.ff01'

]

}

Here are a few points to understand about these data structures:

* The most common and frequently used data structures are lists and dictionaries.
* Tuples never change, which means they can be used for fixed-value types of uses, such as keys for dictionary items.
* Sets are mainly used for testing whether an item is present (inclusion). You can also do operations on sets such as taking the intersection or the union of two sets.
* These data structures can be used together, so that you can have complex data representations, such as lists of dictionaries, dictionaries of lists or lists of tuples.
* 'Comprehensions' are a shorthand way of creating certain data structures. The syntax can be confusing at first but once understood, they can be employed to make your code more concise.
* Copying variable items in Python relates to the previously discussed idea that 'everything is an object'. This lesson will discuss the differences between assignment, copy (a shallow copy), and deep copy.
* **Sequence:** Ordered sequential list of items
* **Heterogeneous:** Items in a list can be of different types
* **Mutable:** Items can be added, removed, inserted, popped, and sorted
* **Items:** List items can be data structures, to create lists of lists, lists of dictionaries or lists of tuples



**Lists Overview**

Lists are one of the most useful tools in the Python language. The following overview describes the basic attributes of a Python list:

* **Sequenced:** Python lists are sequenced, meaning they are ordered. You will therefore be doing operations such as append, insert, and remove.
* **Heterogeneity:** Python lists can hold items of differing types, as shown in the figure; some items are [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address), and one item is actually another list. However, heterogeneity is not required, and can be confusing. So often lists will be of the same type of item.
* **Mutable:** Lists can be changed – that is one of their fundamental purposes. There are operations to append, insert, remove, pop, even to sort a list.
* **Items:** The items in a list can be as simple as numbers or strings, or the items can be data structures such as other lists, tuples, sets, or dictionaries.

Notice from the diagram that square brackets ('[' and ']') indicate lists; when you see data that is listed in Python, if it is in square brackets it is probably a list. The main exception is when you are referring to a specific index for a data structure.

**Creating a List**

* There are two methods you can use to create an empty list, each of which has the same effect. The following two examples show creation of an empty list:
* my\_list = list() # create an empty list

my\_list = [] # create an empty list

* To create a populated list with fixed values, you use square brackets. The following example shows the creation of a populated list:
* # create list holding IP address, username, and password

dev\_info = ['10.3.21.5', 'username', 'password']

**Converting to a List**

You can also take an existing item, such as a tuple or a string, and create lists from those types of data:

Converting a tuple to a list:

device\_tuple = ('10.3.21.5', 'username', 'password')

device\_list = list(device\_tuple)

The resulting data from device\_list would be:

['10.3.21.5', 'username', 'password']

Converting a string to a list:

device\_string = '10.3.21.5 ,username, password'

device\_list = device\_string.split(',')

The example uses the string 'split' function, to separate the string using the character passed into the function (in this case, a comma).

The resulting data from device\_list would be:

['10.3.21.5', 'username', 'password']

**Adding Items to a List**

You can add items to the end of a list using 'append', or you can insert items at a specific location inside the list using 'insert'. Remember that lists are sequenced and so the order of the items may be important.

To append an item to the end of a list:

device\_types = ['IOS','XR'] # create list

device\_types.append('XE') # add 'XE' to list

The result of the append would be the device\_types list as follows:

['IOS','XR','XE']

To insert an item at some location in the list:

device\_types.insert(2,'NX') # insert 'NX' at index 2

The result of the insert would be the device\_types list as follows:

['IOS','XR','NX','XE']

**Removing Items from a List**

There are two ways to remove an item from a list:

* Remove by value: you remove an item by value using the 'remove' function and specifying the value that you want to remove.
* Delete using the specific index by using the **del** operation, and specifying the index of the item you want to remove.

**Note**

Indices start at 0 in Python.

Remove by value:

device\_list.remove('NX') # remove item matching 'NX'

The result of this removal, using the previous 'device\_list' examples:

['IOS','XR','XE']

Remove by index using **del**:

del device\_list[1] # remove item at index 1

**Note**

Remember, Python indexes start at 0. Before the del function is executed, '[IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS)' is at index 0, 'XR' is at index 1, and 'XE' is at index 2.

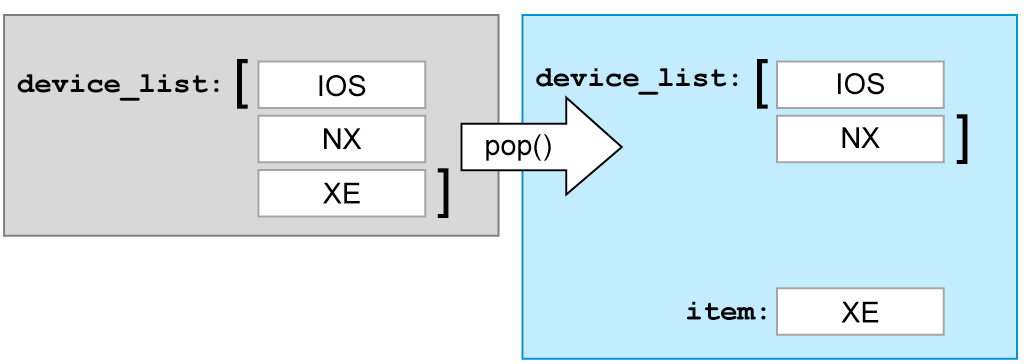
The result of this deletion:

['IOS','XE']

**Popping an Item off a List**

It is also possible to pop an item off a list, where the item is removed from the list but you have a new variable that references the popped item. Popping can occur from the end of the list, from the beginning, or from any point in between.

* Get an item and remove it from a list by index using pop()



Assuming a device\_list with three values 'IOS', 'NX', and 'XE':

item = device\_list.pop() # pop the last item off list

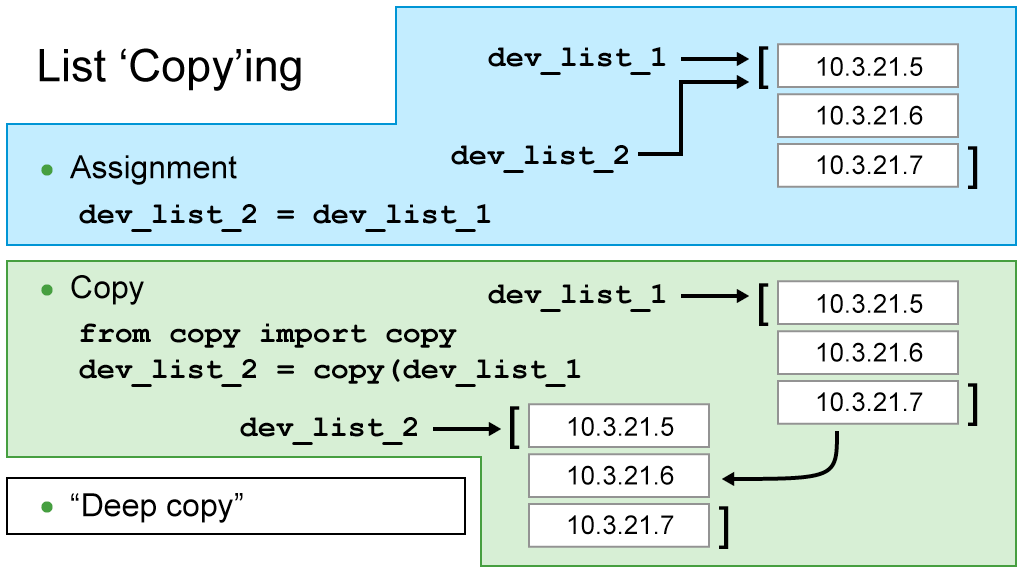
# and point 'item' it it

The resulting value of 'device\_list' will be:

['IOS','NX']

The resulting value of 'item' will be:

'XE'



You will need to understand what is meant by 'assignment' and 'copy' in Python.

## Assignment

Recall that everything is an object, and that variables are just names that reference, or point at, that object. When you do an assignment in Python, you are just having the new variable reference the original value. As you can see in the figure, when you say:

dev\_list\_2 = dev\_list\_1

You are having dev\_list\_2 point to the same list that is pointed to by dev\_list\_1. If you then change the value of items in the list, add items to the list or remove them, both dev\_list\_1 and dev\_list-2 are affected.

## Copy (Shallow Copy)

If you want to make a copy of dev\_list\_1, use the 'copy' function available in Python. In order to use the copy function, you must import it from the Python standard library which is done using the **import** statement.

from copy import copy

dev\_list\_2 = copy(dev\_list\_1)

In this example, you are actually copying the entire list, and all items within it. Using the copy function gives dev\_list\_2 a completely new copy of dev\_list\_1. After a copy, changes to dev\_list\_2 have no effect on dev\_list\_1, and vice versa.

Note that the copy function is actually doing a 'shallow' copy. A shallow copy, in computer programming, means that the only first level of items is copied. If you happened to have a list that had complex data structures such as a list of lists, or a list of dictionaries, a shallow copy does not make a copy of those referenced items—it only copies the first-level items. If you wish to copy all items, no matter how deeply nested are the data structures, you must do a deep copy.

## Copy (Deep Copy)

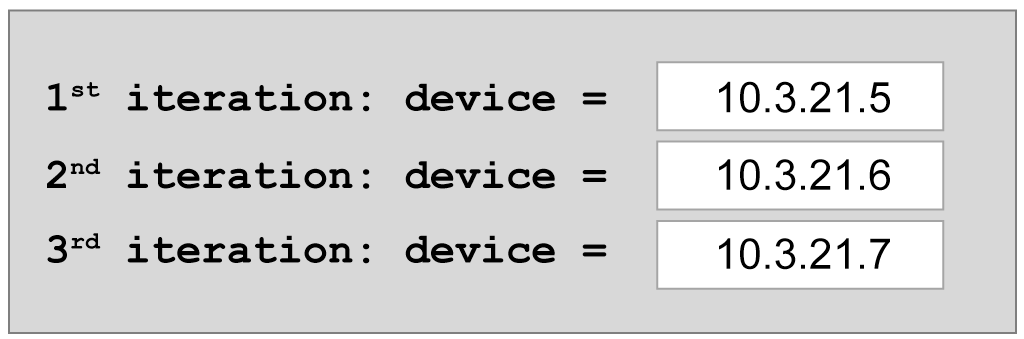
If you need to do a deep copy on your list, use the **deepcopy** function in Python:

from copy import deepcopy

complex\_list\_2 = deepcopy(complex\_list\_1)

The result is that all data from complex\_list\_1 is copied to complex\_list\_2.

One of the main uses of a list is to go through it, either looking for an item, or else taking some actions on each item. This action is referred to as iterating through a list or through a similar sequenced data structure such as a tuple.



Note that control structures for iterating through sequences of things (reading lines in a file, going through items in a list), is covered in full in the module on Control Structures.

Here is an example of iterating through a list:

for device in device\_list:

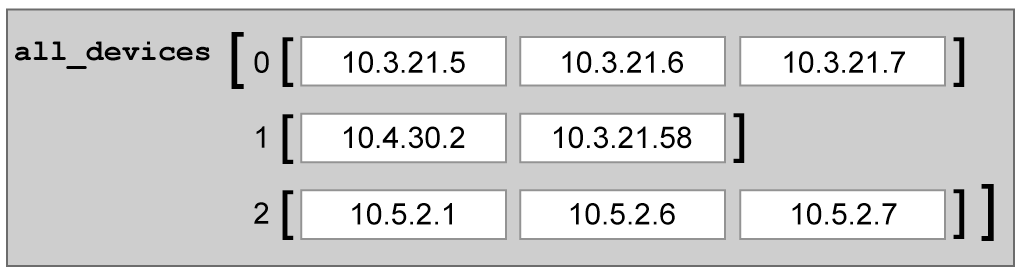
# do something with 'device'

print(device)

In the code above, a **for** statement is used to iterate through the **device\_list**. Spoken in simple English, the statement says "For each item in 'device\_list', let 'device' point to that item, and perform the following actions.

For every iteration, the value of 'device' gets set to the next item in the list. In this way, you sequentially step through each item in the list.

Lists of numbers or strings are interesting, but things get much more useful and practical when they get combined with other data structures.



## List of Lists

Consider a simple list of lists. In the example is a list that is called 'all\_devices', which has been created with the specific purpose of holding the IP address of devices of each device type. List all\_devices[0] represents IOS devices, all\_devices[1] represents NX devices, and so on.

The code to create these lists would look similar to:

ios\_devs = ['10.3.21.5','10.3.21.6','10.3.21.7']

nx\_devs = ['10.4.30.2','10.3.21.58']

xr\_devs = ['10.5.2.1','10.5.2.6','10.5.2.7']

all\_devices = [ios\_devs, nx\_devs, xr\_devs]

And as a result, the data for all\_devices would look like:

[ ['10.3.21.5','10.3.21.6','10.3.21.7'],

['10.4.30.2','10.3.21.58'],

['10.5.2.1','10.5.2.6','10.5.2.7']]

Simple, but powerful as a means of storing handy networking information.

Challenge: What is the result of evaluating the expression **list(list())**?

The correct answer is [], or an empty set. list() creates an empty set and list(list()) creates an empty set of an empty set. For more information, refer to List Basics.

**List comprehensions** are a short way of creating a list, using syntax that is not regularly found in other languages.

Lists are typically created using normal **for** loops or nested **for** loops. Consider an unstructured set of data consisting of a device [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address), username, and a password:

device\_string = ' 1.1.1.1, username , password '

To create a list that does not include the extra white space above, you could do the following:

info\_list = list() # create empty list

device\_info = device\_string.split(,) # create list from device\_string

for item in device\_info:

item = item.strip() # strip white space from device

info\_list.append(item) #add to info\_list

List comprehensions do these two things at once: the **for** statement and the assignment of an item in the list. The trick in identifying a list comprehension is that you see square brackets enclosing what appears to be a **for** statement:

[expression for value in list]

To accomplish the same task using a list comprehension:

device\_string = ' 1.1.1.1, username , password '

info\_list = [ device.strip() for device in device\_string.split(',') ]

Looking at the right-hand side of the statement, the square brackets indicate that this code must be a comprehension of some type. Inside the bracket is the statement:

for item in device\_string.split(',')

This **for** statement will create a list using the split function

device\_string.split(',')

This string manipulation function will take the original string (device\_string), and create a list using the comma (',') as the separator for the list. The list created is:

[' 1.1.1.1', ' username ', ' password ']

The 'split' functionality provides the individual items in the string – IP address, username, password – but a lot of white space remains. It is against this list that the **for** statement operates. So the **for** statement is really:

for item in [' 1.1.1.1', ' username', ' password ']

For every iteration, the value of 'item' will be an item from that list. So the first time through, item will equal:

item = ' 1.1.1.1'

The list comprehension, for the first iteration, looks as follows:

[ item.strip() ]

The **strip** string function strips off white space at the beginning and end of the string. So now, for the first iteration, the list comprehension value will be:

'1.1.1.1'

For the second and third iterations of the **for** loop, the values will be:

'username'

'password'

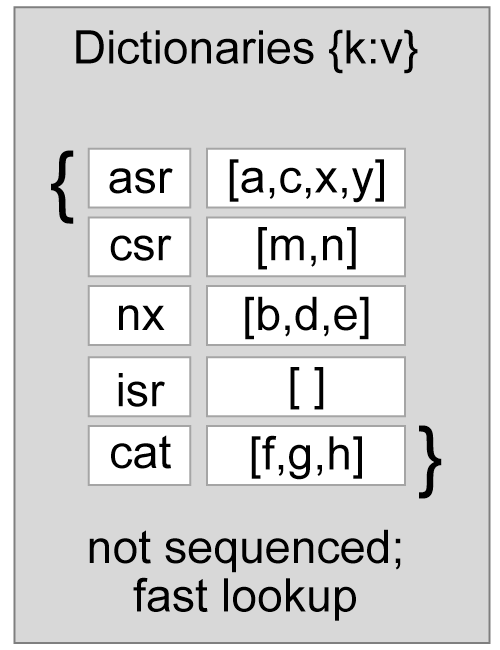
The list that is created by the list comprehension looks like:

['1.1.1.1', 'username', 'password']

This list comprehension was complicated but hopefully one that would be of use to you as a network engineer. Examples of list comprehension abound outside of this training, if you have further interest.

**Dictionary**

* **Unordered:**Not sequenced, accessed by key
* **Keys:** Key must be hashable—string, numeric, or tuple
* **Mutable:** Items can be added, removed, or changed
* **Items:** Items can be simple or complex.



Dictionaries are another extremely useful data structure in Python. A few attributes of dictionaries:

* **Unordered:** Dictionaries are not sequenced; hence they are 'unordered', which means that you don't typically iterate through a dictionary. You go specifically to an individual item using the 'key' value.
* **Keyed:** For every value in a dictionary, there is a key by which it is referenced. These keys must be 'hashable', meaning they must be capable of going through a process to create a 'hash' value for lookup purposes. Strings and numbers are hashable; and because tuples are immutable, they too can be used as keys in a dictionary. Lists, dictionaries, and sets cannot be used as keys.
* **Mutable:** Items in a dictionary can change, hence there are facilities for adding and removing items in a dictionary.
* **Items:** The values that are stored in a dictionary can be simple values or they can be data structures such as lists, other dictionaries, tuples, or sets.

Some characteristics of dictionaries include:

* Python uses the curly braces '{}' to enclose dictionaries.
* Keys can be hashed, as in the figure where Cisco device types are the dictionary keys.
* Values can be single values or they can themselves be data structures – in the figure there are lists for each dictionary item. The intention in the abbreviated example is that the dictionary holds keys for each device type, and the actual value for each key is a list of devices. In the example, the device details are shown as simple letters, but in a real situation they would be IP addresses or the hostnames of the devices.

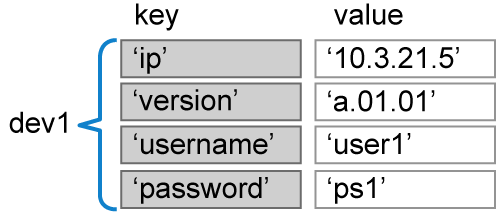
**Creating a Dictionary**

As with lists, there are two methods for creating an empty dictionary. The creation of a populated dictionary involves specifying the keys and values for each item separated by a colon (':').

* Create an empty dictionary
* dev1 = dict () # method 1

dev1 = {} # method 2

* dev1 = {"ip":"10.3.21.5", "version":"A.01.01", "username":"user1","password":"pw1"}



**Creating an Empty Dictionary**

dev1 = dict() # creates empty dictionary "dev1"

dev1 = {} # creates empty dictionary "dev1"

The first example above creates the empty dictionary using the **dict** function. The second example does the same thing, but uses the empty curly braces method.

**Creating a Populated Dictionary**

dev1 = {"ip":"10.3.21.5", "version":"A.01.01", "username":"user1","password":"pw1"}

The example above is different from the dictionary example at the beginning of this lesson. The previous example had Cisco device types as the keys, and the value for each item was a list. In that case, all items where homogeneous, meaning they were of the same type.

In the example above, however, the items are technically all the same 'type', in that they are all strings. However, each item is holding something different –[IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address), firmware version, username, password. In more advanced circumstances, this dictionary would probably not be using a string for IP address, but a class holding the actual binary value. The point however is to understand that dictionaries can be used to hold heterogeneous types of data.

You can see in the example that the data is defined using what are called 'key-value' pairs. The keys and values are separated by a colon, with the keys on the left, and the value on the right. In the following piece of the example:

{"ip":"10.3.21.5", "version":"A.01.01", "username":"user1","password":"pw1"}

The curly braces begin and end the dictionary. The first object that is specified is the key ("ip"); the second object that is specified is the value ("10.3.21.5").

Items in the dictionary are separated by commas.

You will encounter dictionaries that are defined in this way quite often in Python.

You may also notice that the format for defining a dictionary is quite similar to the encoding used with [JSON](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=JSON). You will find that for doing many networking tasks, especially for an [SDN](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SDN)you will end up using both Python dictionaries and JSON-formatted data.

The real value of a dictionary is to immediately find a specific item, by specifying only the key. You used square brackets in lists to reference a specific item by its integer index. You will use key values in the same way for dictionaries.

Consider the following request to get the reference to a specific item in the 'dev1' dictionary that was defined earlier:

ip\_address = dev1['ip'] # get IP address

This request 'finds' the value that is associated with the key 'ip', and returns the value associated with the key. Hence the value now in the ip\_address variable would be:

'10.3.21.5'

Add a new item to the dev1 dictionary:

dev1['OS'] = 'NX-OS' # add new field for 'OS'

Modify an existing item in the dev1 dictionary:

dev1['version'] = 'a.01.12' # modify version number

Deleting items is similarly straightforward:

del dev1['OS'] # delete item located at 'OS'

You may want to test to see if a key is in a dictionary, called a test of 'inclusion'. You can perform this test using the 'in' comparison operator:

if 'OS' **in** dev1:

# take whatever action you desire

The above 'if' statement tests to see if the key '[OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS)' is in the dev1 dictionary. If present, it takes the action specified in the following code block.

Copying dictionaries in Python is the same as copying lists, meaning you should be aware of:

* **Assignment:** Assigning one variable name to reference (point to) another object that always exists. In the example, dev1 and dev2 will reference the same object:

dev2 = dev1

* **Copy:** You can make a copy – a shallow copy – of a dictionary by using the 'copy' function. Unlike lists, you do not need to import the copy function to do a shallow copy of a dictionary.

dev2 = dev1.copy()

* Deep copy: Shallow copies only make copies of the first level of items in a data structure. Doing a deep copy – that copies all levels of items – requires importing and using the **deepcopy** function.
* from copy import deepcopy

dev2 = deepcopy(dev1)

**List of Dictionaries**

Dictionaries become even more useful when you begin to build structures that hold more information by combining dictionaries with other data structures.

Here is a list of dictionaries:

# Create three dictionaries holding device information for specific devices

dev30 = {'name':'xrv-0','ip':'10.1.1.30','user':'cisco'}

dev31 = {'name':'xrv-1','ip':'10.1.1.31','user':'cisco'}

dev32 = {'name':'xrv-2','ip':'10.1.1.32','user':'cisco'}

# Create a list to hold information about all of my devices created above

dev\_list = [i for i in range(3)]

dev\_list[0] = dev30 # sets list item 0 to dictionary dev30

dev\_list[1] = dev31 # sets list item 1 to dictionary dev31

dev\_list[2] = dev32 # sets list item 2 to dictionary dev32

The example above is creating three dictionaries to hold device information, called dev30, dev31, and dev32. The following Python commands are used to create dev\_list:

* **List comprehension:** list comprehension is clearly identified by the square brackets surrounding the for statement. In this situation, a list comprehension is being used to create a dev\_list with some initial empty values.
* **Range:** a functionality in **for** loops, which allows you to easily iterate a specified number of times. In this case, the range command is used to create a list with three empty items at index 0, 1, and 2.

**Note**

Python 'idioms' like the ones above will occur as you read sample code, and the purpose here is to help you become familiar with a few of the common ones.

At this point in the example of a list of dictionaries, there are three dictionaries representing the three devices. The next step is to add these dictionaries to the list, in order to create the list of dictionaries, by assigning the values for dev\_list[0], dev\_list[1], and dev\_list[2] to the dictionaries that were just created:

dev\_list[0] = dev30 # sets list item 0 to dictionary dev30

dev\_list[1] = dev31 # sets list item 1 to dictionary dev31

dev\_list[2] = dev32 # sets list item 2 to dictionary dev32

The result is a list that holds the following data:

[ { 'ip': '10.1.1.30', 'name': 'xrv-0', 'user': 'cisco'},

{ 'ip': '10.1.1.31', 'name': 'xrv-1', 'user': 'cisco'},

{ 'ip': '10.1.1.32', 'name': 'xrv-2', 'user': 'cisco'}]

The above result was printed by Python, using the 'pretty printing' function available, which is discussed later in this section.

Instead of creating a list with 3 items and then assigning the dictionaries to those items, you could have created an empty set and used the append function to add the dictionaries to the list. The code would look similar to:

dev\_list = []

dev\_list.append(dev30) # sets list item 0 to dictionary dev30

dev\_list.append(dev31) # sets list item 1 to dictionary dev31

dev\_list.append(dev32) # sets list item 2 to dictionary dev32

You will often be able to accomplish the same task multiple ways. Sometimes the choice is a matter of taste or personal preference, but as you become more proficient you will also learn how different Python methods impact performance or reusability of your code.

The choice is often up to the developer, but it is important to strive to be consistent with pre-existing coding styles, if updating or modifying existing code.

**Dictionary of Dictionaries**

You will sometimes wish to create dictionaries of dictionaries. This example is creating the same functionality as before, only this time using a different data structure. Here is the code:

dev30 = {'ip':'10.1.1.30','user':'cisco'}

dev31 = {'ip':'10.1.1.31','user':'cisco'}

dev32 = {'ip':'10.1.1.32','user':'cisco'}

devices = {}

devices['xrv-0'] = dev30 # sets item 'xrv-0' to dev30

devices['xrv-1'] = dev31 # sets item 'xrv-1' to dev31

devices['xrv-2'] = dev32 # sets item 'xrv-2' to dev32

In the example above, the same dev30, dev31, and dev32 dictionaries are created to hold device information. Then a dictionary called 'devices' is created. The final step is adding the individual device dictionaries to the dictionary called devices.

The outer dictionary however provides some extra value. Notice that the keys being used are actually the names for the specific devices. So the dictionary holds extra information, that wasn't present in the list of dictionaries.

The actual data looks as follows:

{ 'xrv-0': { 'ip': '10.1.1.30', 'name': 'xrv-0', 'user': 'cisco'},

'xrv-1': { 'ip': '10.1.1.31', 'name': 'xrv-1', 'user': 'cisco'},

'xrv-2': { 'ip': '10.1.1.32', 'name': 'xrv-2', 'user': 'cisco'}}

**Dictionary of Lists of Dictionaries**

The final example features three levels of hierarchy, this time being a dictionary of lists of dictionaries.

The example demonstrates a use case where there are several devices in the network using various OS types. The goal is to keep track of these devices. The solution features the following:

* **Device Types dictionary:** An outer dictionary, with items representing the different types of devices (XR, NX, [IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS)).
* **Devices list:** For each device type in the dictionary, there is a list of devices from the network that are that specific type (XR or NX or IOS).
* **Device Info dictionary:** For each device in the list, there is a dictionary containing the device information for that specific device.

The sample code should look familiar – roughly the same as the list of dictionaries in the first example. Notice that the list is specific to a particular device type. And an extra dictionary has been added which contains the devices sorted by device type.

Here is the code:

# Create dictionaries holding device information for these three XRv devices

dev30 = {'name':'xrv-0','ip':'10.1.1.30','user':'cisco'}

dev31 = {'name':'xrv-1','ip':'10.1.1.31','user':'cisco'}

dev32 = {'name':'xrv-2','ip':'10.1.1.32','user':'cisco'}

# Create a list for our XRv devices that we have just defined

xrv\_devices = []

xrv\_devices.append(dev30)

xrv\_devices.append(dev31)

xrv\_devices.append(dev32)

all\_devices = {'xrv':xrv\_devices, # all XRv devices go here

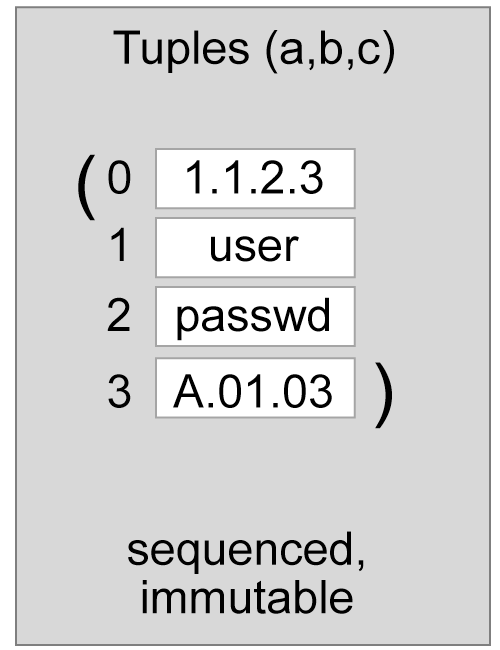
'nx-os':nx\_devices, # all NX devices go here

'ios':ios\_devices} # all IOS devices go here

Only XRv devices are shown in the example, and so you should assume that 'nx\_devices' and 'ios\_devices' have been defined elsewhere.

**Tuples**

* **Sequence:** Ordered collection of items
* **Heterogeneity:** Items can be of different types
* **Immutable:** Cannot be changed
* **Items:** Items can be strings or numbers or can be data structures such as lists or dictionaries



**Tuples Overview**

Tuples are very much like lists, with a few significant differences. The following are attributes of tuples:

* **Sequence:** Tuples, like lists, are ordered sequences of items which means that you reference items in the tuple using the numerical index.
* **Heterogeneity:** Tuples, like lists, can have items of differing types.
* **Immutable:** Unlike lists, tuples cannot be changed. Once you have created a tuple, the values set can never be changed. If your tuple needs to change for some reason, create a new tuple.
* **Items:** Items in a tuple can be simple, like strings or numbers, or they can be data structures of some type, like lists or dictionaries.

Tuples are identified in Python by parentheses, '(' and ')'.

**Comparing Tuples and Lists**

* Tuples are more efficient than lists
* Tuples cannot be accidentally changed
* Tuples are actually used by **Python for passing parameters to functions**
* **Named tuples are sometimes a better way to use tuple data structures**, allowing you to refer to the items in a tuple by their name, rather than by the numerical index.
* Since tuples never change, if they consist of immutable objects like strings and numbers, they can be used as dictionary keys.

To summarize, tuples are less flexible than lists, but they do have several distinct advantages, that may be useful in certain situations.

**Creating a Tuple**

Creating a tuple is like creating a list. Empty tuples can be created using either the tuple function, or by using round brackets:

my\_tuple = tuple() # creates an empty tuple 'my\_tuple'

my\_tuple = () # creates an empty tuple 'my\_tuple'

However, creating an empty tuple has limited value, since items cannot be added to it. Of more interest is creating a populated tuple. The following code shows some examples of creating populated tuples.

The following example creates a tuple, which looks just like creating a list except for the parenthesis.

# create tuple 'dev\_info' holding IP, username, and pw

dev\_info=('1.1.1.1','username','password')

The following example creates the same tuple, but notice that the parenthesis are not required. Omitting the parenthesis is sometimes called **'tuple packing'** – Python implicitly knows from the statement syntax that a tuple is being created.

# create tuple, no '()' required

dev\_info='1.1.1.1','username','password'

You can create tuples with values that are data structures. In the following example, a tuple with three dictionary items is created:

# Create three dictionaries holding device information for specific devices

dev30 = {'name':'xrv-0','ip':'10.1.1.30','user':'cisco'}

dev31 = {'name':'xrv-1','ip':'10.1.1.31','user':'cisco'}

dev32 = {'name':'xrv-2','ip':'10.1.1.32','user':'cisco'}

# create tuple 'devices' with three device dictionaries

devices=(dev30, dev31, dev32)

**Concatenating a Tuple**

You can create other data structures and variables from tuples as well. The following example takes a tuple, and creates a list:

device\_tuple = ('1.1.1.1','username','password')

device\_list = list(device\_tuple)

The inverse of tuple packing, **'tuple unpacking**, is also possible. In this situation, a tuple is separated into its constituent parts. In the following example, a tuple of device information is used to set individual variables of 'ip', 'user', and 'pw':

device\_tuple = ('1.1.1.1','username','password')

ip, user, pw = device\_tuple

**Named tuples** can provide significant improvements in code readability. Tuples suffer from the fact that in certain cases, the items that are stored are not a collection of the same type of item, but are actually different types of items.

In such cases, it would be helpful to be able to refer to specific items in the tuple by name, rather than by index. For example, with the device information tuple, it would be nice to refer the IP address by name, the username by name, and the password by name, rather than by index.

The following example shows how named tuples could be used to improve code readability. Consider the indexed way of doing things. For the dev\_info tuple, the version field is accessed by using the numerical index of its location in the tuple, which is 1:

dev\_info = ('iosxrv1', 'A.01.01')

version = dev\_info[1]

Contrast the example with using a named tuple. The setup takes some extra code, but referencing the field within dev\_info is much cleaner:

from collections import namedtuple

Info = namedtuple('**Info**',***'name version'***)

dev\_info = Info(name='iosxrv1', version='A.01.01')

version = dev\_info.version

To use a named tuple, you first declare that you are using named tuple by importing it. Then you **create an object class** for your device information, in which you define the fields (name and versions).

Creating the tuple itself involves declaring your named tuple class 'Info' and specifying the values for the named fields, 'name' and 'version', which are set to 'iosxrv1' and 'A.01.01' respectively.

At this point, your tuple is specified using the actual name for the field, which is the version.

====================== lab code of namedtuple ================

from collection import namedtuple

from pprint import pprint

Dev\_info = namedtuple('Dev\_info', ['name', 'os\_type', 'ip', 'user', 'password'])

os\_types = set()

file = open('devices', 'r')

for line in file:

device\_info = Dev\_info(\*(line.strip().split(',')))

print 'Device Infomation: ', device\_info

if device\_info.os\_type not in os\_types:

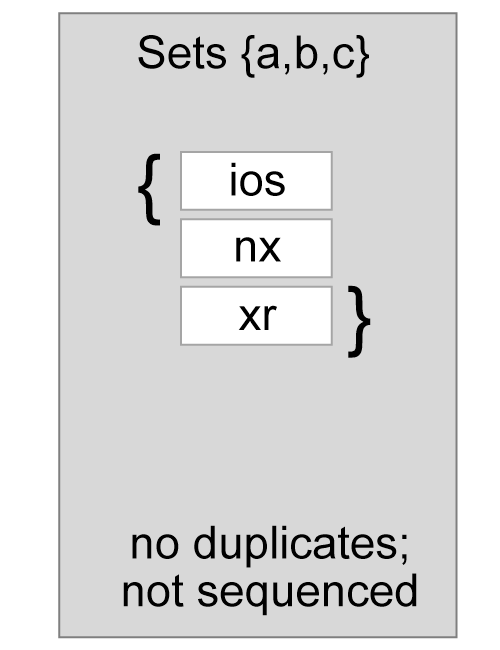
os\_types.add(device\_info.os\_type)

file.close()

pprint(os\_types)

7.13 Sets

* **Unordered:** Unordered collection of items
* **Unique Members:** No duplicate members allowed in a set
* **Hashable:** Members must be hashable—numbers, strings, or tuples (no dictionaries or lists)
* **{ }:**Uses curly braces like dictionary but with a single value for each member (no key)
* **Mutable:** Sets can be modified



**Sets Overview**

Sets are data structures that are best used in **testing for the presence or absence of an item, and for performing operations such as unions and intersections**. Some characteristics of sets:

* **Unordered:** Sets contain items in no particular order or sequence – the only consideration for a set is whether an item is 'in' the set or not.
* **Unique items**: There are no duplicate items in a set, every item is present only once. If you take a list with duplicate items, and coerce it into a set, all duplicates will be removed.
* **Hashable:** Members of a set must be hashable, meaning that they must be either simple, like numbers or strings, or tuples, which themselves are immutable and hence can be hashed. This requirement makes sense when you consider the operations that are performed on sets, such as testing for inclusion, are easily done with hashable items, but would be quite difficult and inefficient otherwise.
* **Curly braces {}:** You may be confused by the fact that sets are enclosed in curly braces, as with dictionaries. However, consider that dictionaries are populated with key-value pairs, while sets are populated with single items, which eliminates confusion and makes the two curly braces data types easy to distinguish.
* **Mutable**: You can add and remove items from a set.

**Creating a Set**

Unlike lists and dictionaries, there is only one way to create an empty set:

dev\_os\_types = set()

There is no 'empty braces' notation as is possible with lists and dictionaries. The reason is because the type of braces that are used – curly braces {} – are already used by dictionaries, and so using them to create an empty set would be ambiguous.

Creating a populated set is similar to creating populated lists, dictionaries, and tuples: include the items inside the braces. Notice that these items can be distinguished from items in a dictionary, since they are individual, not key-value pairs. Hence Python is able to know whether you were creating a set or a dictionary. The example below shows creating a populated set:

dev\_os\_types = {'ios','xr','nx'}

It is also possible to create a set from a list. The general format is to use the 'set' function, as in the following example. Consider a list that holds the device types of all devices that have been discovered in the network. This list will have many items, with many duplicates. Creating a set of device types from that list, is straightforward:

devices\_list = ['xr','nx','nx','xe','xr','nx','xr','ios','nx','xe','xr','xr','nx']

dev\_\_os\_types = **set**(device\_types\_list)

if 'ios' in dev\_os\_types:

# do something with ios devices

A more complicated example is shown in the figure. Imagine several device tuples, these being dev30, dev31, dev32, and so on, which are actually named tuples, allowing easy reference to the [OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS)type of each device by using the '.type' suffix. In the example, the **devices\_list** list is created from the 'type' field of these device tuple objects. Then this list is used to create the list of all 'dev\_os\_types' present in the network.

devices\_list = [dev30.type, dev31.type, dev32.type, dev33.type, dev34.type, dev35.type]

dev\_os\_types = set(devices\_list) # create set of types from list of all device's OS types

if 'ios' in dev\_os\_types:

# do something with ios devices

Sets can easily be created from data structures such as lists, allowing for easy testing for inclusion, and other set-type operations if so desired.

**Modifying Sets**

You can add items to a set using the 'add' function:

dev\_types = set()

dev\_types.add('crs') # add device type 'crs' to set

You can update items to a set as well – **update is really just adding more than one item at a time**.

dev\_types.update(['isr','asr']) # add more device types

Since items are hashable, it is possible to remove items by specifying the value:

dev\_types.remove('crs') # remove device type 'crs'

**Testing Sets**

One of the main reasons for having sets is to test for inclusion, given some other value. For example, if there is a list of used[IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address) for the network, it may be necessary to test to see if a given IP address has already been used. The following example shows the testing of inclusion for this situation:

ip = '10.3.21.17'

used\_ip\_addresses = {'10.3.21.1','10.3.21.2','10.3.21.3', '10.2.21.4','10.3.21.5'}

if ip in used\_ip\_addresses # test if 'ip' is used yet

In addition to these types of tests, operations such as finding the intersection or union of two sets is also possible.

7.14 Printing Data Structures

You will likely want to view your data structures as you develop your code, and also when you wish to print the results of your application. Manually doing so would be difficult, so Python provides a library specifically for that purpose. The library is called 'Pretty Print', or **pprint**.

Formatting output is quite simple, and there are multiple options that can be tailored to your desires. The outputs from the last few dictionary examples were formatted with the **pprint** library. A simple example is given here:

from pprint import pprint

... # create your data

pprint( your\_data\_structure )

The first step is to import the **pprint** function from the **pprint** library. That done, all you need to do is declare use the **pprint**function to print your data.

Section 8: Comparing Network Information

8.1 Introduction

With network scripting, it is common to want to compare two pieces of information. For example, you may create a script that checks the version of IOS software running on every router in the network and generates a list of routers running a specific IOS version. This script could be used to quickly identify all the routers running a specific version of IOS so they can be scheduled for upgrades.

To accomplish this task, your script will log in to each router, issue a **show version** command, and process the output.

r1#**show version**

Cisco IOS Software, IOSv Software (VIOS-ADVENTERPRISEK9-M), Version 15.4(2)T1, RELEASE SOFTWARE (fc3)

Technical Support: http://www.cisco.com/techsupport

Copyright (c) 1986-2014 by Cisco Systems, Inc.

Compiled Thu 26-Jun-14 15:58 by prod\_rel\_team

The first thing your script needs to do is process each line of the **show version** output to find the version number, which in this example is 15.4(2)T1. In order to find this string of characters within the output, you will use a type of comparison match called a regular expression. **Regular expressions** are used to match a specific pattern of characters in a string of characters. The code to do this match would be:

version\_pattern = **re**.**compile**('Version ([0-9]\*.[0-9]\*\([0-9]\))')

version = version\_pattern.search(version\_output)

running\_version = version.group(1)

Once you have extracted the version number and stored it in the variable running\_version, you will need to compare it against the IOS version number you are looking for. For this, you’ll use a simple comparison match. If the IOS version number you’re looking for is stored in the variable target\_version and the IOS version number of the router you just issued the ‘show version’ command on is stored in the variable running\_version, the simple comparison would be:

if target\_version == running\_version

The == symbol compares the two variables as strings and returns a value of true if they match.

In this lesson, you will learn about the various ways that Python objects can be compared and evaluated against each other.

Python provides the traditional comparison operators that are used in many different contexts, including equal ('=='), not equal ('!='), less than ('<'), greater than ('>'), less than or equal to ('<='), greater than or equal to ('>=').

You can perform arithmetic comparisons on numbers using the comparison operators ==, !=, <, >, <=, and >=. String comparisons are performed lexicographically, basically meaning in dictionary or alphabetical order.

Python also allows for testing of membership, using the **in** operator. The most obvious example would be for testing membership in a set. Less obvious is the fact that you can also test for membership in a list, tuple, or as a key in a dictionary. For strings, **in** tests for the presence of a substring within a string.

Lists and tuples are compared item by item. Dictionaries have their keys sorted, then are compared for both key and value.

Comparisons for sets use the membership operator 'in'.

8.2 Simple Comparisons

Simple comparisons refers to comparisons of basic items such as strings and numbers:

* Equality or inequality of strings or numbers. The examples compare the device version with some expected version number to see if it needs to be updated:
* if version == dev\_info.version:

if version != dev\_info.version:

* Arithmetic comparison of items to provide equality or greater/less than evaluation. The example checks to see if the calculated utilization for a link is greater than the maximum:

if utilization > max\_utilization:

* Set comparison involves testing if a specific value is a member of the set, or whether it is an item in a list or tuple, or whether it is a key in a dictionary. The example tests if the device type is a Cisco device type:

if dev\_info.os\_type in cisco\_os\_types:

8.3 Complex Comparisons

There are certain comparisons that would be beneficial in a networking environment, but are impractical using basic comparison methods. For example:

* **Versions:** Version strings can vary widely, and therefore it is not possible to do a straight string comparison to determine if one version is less than, or greater than, the other. For example, some version strings have a major version and a minor version that are separated by dots; some have major, minor, and build number, and so on.
* **Addresses:** [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address) cannot be compared lexicographically either—an IP address of 198.1.1.1 will evaluate as less than IP address 2.1.1.1.

There are two approaches that can be used to handle complex comparisons. The two approaches can be used together or separately.

* **Decompose:** One strategy is to take an otherwise complex string, and separate it into its constituent parts. For example, with a version number, the example shows splitting the version string into a list, using the dot '.' to separate the items. In this way, it would be possible to examine major versions against each other, then the minor versions.

version\_list = dev\_info.version.split('.')

* **Normalize:** Another strategy would be to take the item in question and put it into a normalized form. A good example would be for dealing with[MAC address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=MAC%20address), which can be presented in [CLI](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CLI)s with colons (':'), or hyphens ('-') every two bytes. Sometimes they are even represented with dots ('.') every four bytes. Normalizing by changing all characters to lower case, then removing the hyphens, colons, or dots, helps to make all MAC address forms 'equal', so that they can be compared against one another, no matter where they came from.
* mac\_addr = mac\_addr.lower() # get mac address into lower case
* mac\_addr = mac\_addr.replace(':','') # strip out colons
* mac\_addr = mac\_addr.replace('-','') # strip out dashes

mac\_addr = mac\_addr.replace('.','') # strip out dots

The result is that the mac address has had all the special characters removed, which is good for computer programs and their ability to compare data without confusing and inconsistent visual aids.

8.4 Regular Expressions

In Python, in order to use regular expressions, you will do the following:

* Import **re**, the regular expression library
* Define and compile your regular expression match pattern, which will have wildcards, possible values for matching, number of repetitions, groupings, and so pm, as in the examples above for matching [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address) and [MAC address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=MAC%20address) addresses.
* Perform a search for your specific string, using the pattern created above
* The result is a regular expression match object holding the actual matched value from the string.

## Regular Expression Example

The following code provides an example of using regular expressions to find and extract the software version from the output of a Cisco [IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS)XR device.

The actual output of the command to show the version, in brief format, is as follows:

RP/0/0/CPU0:kcy#**show version brief**

Thu Dec 3 16:11:02.098 UTC

Cisco IOS XR Software, Version 5.3.1.24I[Default]

Copyright (c) 2015 by Cisco Systems, Inc.

ROM: GRUB, Version 1.99(0), DEV RELEASE

kcy uptime is 1 day, 21 hours, 27 minutes

System image file is "bootflash:disk0/xrvr-os-mbi-5.3.1.24I/mbixrvr-rp.vm"

cisco IOS XRv Series (Pentium Celeron Stepping 3) processor with 3169911K bytes of memory.

Pentium Celeron Stepping 3 processor at 2718MHz, Revision 2.174

IOS XRv Chassis

1 Management Ethernet

7 GigabitEthernet

97070k bytes of non-volatile configuration memory.

866M bytes of hard disk.

2321392k bytes of disk0: (Sector size 512 bytes).

You can see the version number on the third line of output:

Cisco IOS XR Software, Version 5.3.1.24I[Default]

The goal is to extract the relevant parts of that version string, so that you can do some comparisons against a desired version number. The steps to accomplish this task are:

* Create a regular expression pattern that will be used to match the important parts of the version string. In this case, that pattern will match the string 'Version ' followed by three dot-separated numbers.

version\_pattern = re.compile('Version ([0-9]\*\.[0-9]\*\.[0-9])')

In this case, the version string is relatively straightforward; other devices and variants may require a more involved pattern match string.

* Perform a search for the pattern in the output of the **show version brief** command that was issued. Note that there are two ways to perform the search. You can use the 're' object like in the 'compile' step previously:

version = re.search(version\_pattern, version\_output)

You could also use the 'version\_pattern' object directly to perform the search:

version = version\_pattern.search(version\_output)

Each method yields the same result, a 'version' object that contains specifics about the match that was made. With Python, there are often multiple ways to create your code. Your choice may depend on performance or perhaps even personal preference.

* Your regular expression search produces matched text from the input string, which is organized into groups. Roughly speaking, groups are defined by parenthesis. In the example, group 0 is the entire matched string; and group 1 is the part that was matched inside the inner parenthesis:

version\_string = version.group(1) # get the version part

The code sets variable 'version\_string' to a subset of the matched string – and that subset is just the purely numerical portion of the matched version string. In this case, that would be the value '5.3.1'.

* The resulting version string '5.3.1' is ready to be decomposed into its version parts, for the major version, minor version, and so on. With string manipulation it is easy to create a list of the different numerical version numbers:

version\_numbers = version\_string.split('.')

In this example, you have seen how regular expressions can be useful in taking complex [CLI](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CLI) output and extracting the important pieces of information. That information can be used in comparisons, and those comparisons can be used to create code that evaluates data and takes the appropriate action..

8.5 Creating Comparisons

The code block includes defining the regular expression that will be used to find the management IP address.

import re

# Print heading

print ''

print 'Devices and their Management IP addresses'

print '=================================='

# Create regular expression to find the Mgmt IP address

**ip\_addr\_pattern = re.compile('Mgmt:([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})')**

# Read all lines of device information from file

file = open('devices','r')

for line in file:

device\_info\_list = line.strip().split(',')

**mgmt\_addr = ip\_addr\_pattern.search(line)**

**device\_info['ip'] = mgmt\_addr.group(1)**

print ' Device:', device\_info['name'], ' Mgmt IP:', device\_info['ip']

print '' # Print final blank line

file.close() # Close the file since we are done with it

# Section 9  Conditional Code

# 9.1 Introduction

When creating networking scripts, you will find that you almost always use comparisons such as >, <, == and regular expressions as part of a conditional statement such as IF-THEN-ELSE. When you are comparing two pieces of information, such as the version of IOS running on a router with a target version of IOS, you are typically making the comparison for the sole purpose of taking different actions based on the results of the comparison.

The primary conditional code statement in Python is the **if** statement.

There are three possible components of an **if** statement:

* **if**: Evaluates a comparison operation and if true, executes the associated code block.
* **else**: If the previous **if** comparison returned false, executes the associated **else** code block.
* **elif**: Combines an **else** statement with a subsequent **if** statement, which includes a new comparison associated with this new **if** clause.

If you are writing a script that checks the running version of IOS on all routers in the network, you may want the script to provide a list of routers that are running an IOS version that matches the one you specify.

if running\_version == target\_version:

# alert user that this router is running the version of IOS we are looking for

else:

# the versions do not match, so check the next router

If you are trying to determine if the version of IOS running on a router, 15.4(2), is older than the version you are looking for, 15.5, you will need to perform a multi-step, complex comparison by chaining multiple IF statements together.

target\_version\_major = 15

target\_version\_minor = 5

running\_version\_major = 15

running\_version\_minor = 4

if running\_version\_major < target\_version\_major:

# alert the user that this router is running an older IOS version

elif running\_version\_major > target\_version\_major:

# do nothing since router is a running a newer version

elif running\_version\_major == target\_version\_major:

# we need to compare the minor versions

If the major version numbers are the same, you will likely want to make use of a nested **if** statement by continuing the script with the following code:

if running\_version\_minor < target\_version\_minor:

# alert the user that this router is running an older IOS version

elif running\_version\_minor > target\_version\_minor:

# do nothing since this router is running a newer version

else:

# do nothing because this router must be running the same version

You will notice that the last statement used an **else** conditional. If the first two comparisons returned false, the two minor version numbers must match exactly.

# 9.2 Conditional Code: IF

A simple **if** statement involves only the **if** itself, followed by the condition, followed by a code block that will get executed if the condition evaluates to true. Networking use cases of **if** statements include:

* Comparing if the software version of a device is the correct value. For example, equal to the current version on which all devices should be running.
* # Check if the software version is current

if version == current\_version:

This example would be useful for a device software management application attempting to make sure that all devices in the network are kept up to date on the correct version of software.

* Comparing if the last time a device status was checked is greater than the configured status interval.
* # Check if need to check status

if last\_check[device] > status\_interval:

This example would be useful for a network management application that attempts to maintain reasonably current status for all devices in the network.

* Comparing if the current [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address) is valid.
* # Check if the IP address is known

if ip\_address in known\_ip\_addresses:

This example is a membership test which checks the IP address against the set 'known\_ip\_addresses'. This type of test would be useful in an IP address diagnostic application, where an event has detected some type of issue with a particular IP address, and the application is looking to see if there is more information related to this IP address (for example, the user who is currently associated with this IP address).

This type of test may also be useful in an IP address management application, where IP addresses would be tested against a pool of unused IP addresses.

With Python, it is also possible to test against whether the data structure is empty or not. The following examples show testing for empty:

* Checking the list of available IP addresses, to see if anything is left in the pool.
* # Check there are available IP addresses

if available\_ip\_addresses:

* Checking to see if there are any devices in the dictionary which could be useful when determining if a discovery process has yielded any positive results.
* # Check if there are any devices in the dictionary

if device\_dictionary:

* Checking to see if a regular expression searching for a version string has been successful or not.
* # Check if regex found the desired version pattern

if version\_pattern.search(version\_output):

# 9.3 Conditional Code: ELSE and ELIF

It is quite possible to write an **if** statement which is complete in and of itself. For example, "if this condition is met, perform this operation." However, much of the time you will want to have an **else** clause associated with the **if.**

The general form of the **else** statement is:

if condition:

# code for when condition is true

else:

# code for when condition is false

Like **for** loops, the beginning and end of the code block for the **else** statement is identified by indentation; when the indentation goes back to the original level, the **else** code block is complete. The following example shows a simple string comparison of the version, with code that prints whether the code is up to date, or whether it needs to be updated.

if version == current\_version:

print(dev\_info.name + ': version is up-to-date')

else:

print(dev\_info.name + ': ' + dev\_info.version)

print(dev\_info.name + ': version needs updating')

## elif

Sometimes your logic will call for a sequence of comparisons, each requiring a different action. Some languages have **switch** or **case** statements; Python uses a condensed version of an 'else if' statement, called **elif**.

An example of the use of **elif** would be testing for the [OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS) type of the device to which you are talking, and taking different actions based on that information:

if device.os\_type == 'ios':

# perform actions appropriate for ios type devices

elif device.os\_type == 'xr':

# perform actions appropriate for xr type devices

elif device.os\_type == 'nx':

# perform actions appropriate for nx type devices

elif device.os\_type == 'xe':

# perform actions appropriate for xe type devices

else:

# Unknown device type, take appropriate action

# 9.4 Conditional Code: Advanced if Statements

## Inline if Statements

There may be situations in which the code blocks for your **if** statements are only one statement. In such cases, it can be useful to condense your code into an inline **if** statement. In these situations, the code begins with an assignment statement, followed by the **if** condition, followed by the **else** clause. The format in the longer case is:

if condition:

x = y

else:

x = z

The inline form is:

x = y if condition else z # conditional assignment

As you can see, this example is a more compact version of the standard **if-else** statement. Once the structure is understood, the inline **if** statement provides a simple way to condense four Python statements into one. A real-world example is:

# Set flag if update is needed

update = true if version != current\_version else false

The example is testing to see if the software version of a device is equal to the current version; if not, the 'update' flag is set to true, so that a report of non-compliant devices can be generated.

## Nested if Statements

Sometimes it is necessary to test for multiple conditions, in order to make a decision. To support these use cases, Python allows for the nesting of if statements.

However, it is important to understand the dangers in creating code that does this type of nesting—or sometimes, even more nesting. The issue has to do with correctly creating the **else** statements with the **if** statements to which they correspond. When you consider that some **if** statements can have no else statements, as in the example below, it can get confusing to untangle the logic of such constructs.

if condition-1:

# code for when condition-1 is true

if condition-2:

# code for when condition-2 is true

else:

# which 'if' is this the 'else' for?

# indentation provides the answer

Some languages rely on curly braces or **begin-end** or just **end** to identify specific code blocks. Python does not use these constructs, but relies on indentation which makes it easier to visually see the beginning and end of code blocks to insure that **if-else** statements are correctly structured.

if version.major == current\_major\_version:

if version.minor == current\_minor\_version:

print(dev\_info.name + ': version is up-to-date')

else:

print(dev\_info.name + ': minor version is not current!')

else:

print(dev\_info.name + ': major version is not current!')

You should still take care when creating nested **if** statements, however, as they can be difficult to logically follow, even with enforced indentation as in Python. Sometimes it will be preferable to use multiple conditions on the same **if** statement.

## Nested if Statement Alternative

As an alternative to nested **if** statements, consider putting multiple conditions in the initial **if** statement, where appropriate.

Python provides the Boolean operations **and**, **or**, and **not**. The **not** operation works on a single item, but **and** and **or** are useful for combining comparison values on a single **if** statement.

if condition-1 and condition-2:

# code for when both conditions are true

else:

# code for when either condition is false

The nested version check example that was shown previously could be written as follows.

if (version.major == current\_major\_version) and (version.minor == current\_minor\_version):

print(dev\_info.name + ': version is up-to-date')

else:

print(dev\_info.name + ': version is not current!')

If it was desired to highlight whether the major or minor version was out-of-date, the following code could be used:

if (version.major == current\_major\_version) and (version.minor == current\_minor\_version):

print(dev\_info.name + ': version is up-to-date')

elif version.major != current\_major\_version):

print(dev\_info.name + ': major version is not current!')

elif version.minor != current\_minor\_version:

print(dev\_info.name + ': only the minor version is not current!')

There are multiple ways to construct this logic; however, the purpose of this simple example is to show that grouping conditions together in the same **if** statement can make the code more readable, easier to understand, and debug.

## Precedence

You can combine as many conditionals as you desire, using **and** and **or**. In order to organize the evaluation of your conditional, you will need to understand precedence rules, or explicitly specify precedence using parenthesis.

Precedence refers to the order in which Python will evaluate sets of comparisons in a conditional statement.

A complete table of precedence can be found in many places online; a simplified version is as follows

* Arithmetic comparison operators have equal precedence, and are evaluated left-to-right.
* Boolean operator precedence is, in order: **not**, **and**, **or**.
* Bitwise operators (**OR**, [XOR](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=XOR), **AND**) are higher precedence than Boolean operators.

The good news is you do not need to memorize the table of precedence, because parentheses can be used to explicitly instruct Python what is your preferred order of evaluation. Parentheses are safest because they override any implicit precedence rules.

# Lab

# Using Conditionals with Network Devices

In the exercises, you will use conditionals to perform typical network tasks.

The lab is ready!

Launch

## Count Routes Per Interface

In this exercise, you will be using a text file that has information from a **show ip route** command. The file is **~/Desktop/PRNE/section09/ip-routes** and contains data that looks like:

Codes: C - connected, S - static, R - RIP, B - BGP, (>) - Diversion path

D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area

N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type

C 10.11.13.0/24 is directly connected, 3d05h, MgmtEth0/0/CPU0/0

L 10.11.13.4/32 is directly connected, 3d05h, MgmtEth0/0/CPU0/0

S 10.16.0.0/16 [1/0] via 198.18.1.1, 3d05h

i L2 11.11.1.0/24 [115/10] via 58.0.0.21, 2d10h, GigabitEthernet0/0/0/0

S 11.11.2.0/24 is directly connected, 3d05h, Null0

i L2 11.11.3.0/24 [115/20] via 58.0.0.21, 2d10h, GigabitEthernet0/0/0/0

[115/20] via 49.0.0.30, 2d10h, GigabitEthernet0/0/0/2

i L2 11.11.4.0/24 [115/30] via 49.0.0.30, 2d10h, GigabitEthernet0/0/0/2

i L2 11.11.5.0/24 [115/10] via 48.0.0.27, 3d05h, GigabitEthernet0/0/0/1

You will count how many routes are associated with each GigabitEthernet port.

Note: You will be able to use several components of this lesson in this lab, in particular:

* Using an **if** statement to test for 'empty'
* Using a nested **if** statement
* Using an **else** statement
* Using an inline **if** statement

**Step 1**

Create a regular expression pattern to match the 'GigabitEthernetn/n/n/n' string.

### **Answer**

|  |
| --- |
| from pprint import pprint  import re  # Create regular expression to match Gigabit interface names  gig\_pattern = re.compile('(GigabitEthernet)([0-9]\/[0-9]\/[0-9]\/[0-9])') |

**Step 2**

Create a dictionary to hold the count of routes that will get forwarded out each interface.

### **Answer**

|  |
| --- |
| routes = {} |

**Step 3**

Read the route information from the text file **~/Desktop/PRNE/section09/ip-route**.

### **Answer**

|  |
| --- |
| # Read all lines of IP routing information  file = open('ip-routes','r')  for line in file: |

**Note**

Make sure you are in the **~/Desktop/PRNE/section09/** directory before running your script. This is because the **ip-route** file the script is attempting to open is in current working directory.

**Step 4**

If there is a match, add one to the count of routes for that specific interface.

### **Answer**

|  |
| --- |
| match = gig\_pattern.search( line ) # Match for Gigabit Ethernet  # Check to see if we matched the Gig Ethernet string  if match:  intf = match.group(2) # get the interface from the match  routes[intf] = routes[intf]+1 if intf in routes else 1 |
|  |

**Step 5**

When all lines have been read, print the counts of [IP](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP) routes that are getting forwarded out each interface.

### **Answer**

|  |
| --- |
| print ''  print 'Number of routes per interface'  print '------------------------------'  pprint(routes)  print '' # Print final blank line |

**Step 6**

Run your application and verify the output.

### **Answer**

Your output should look similar to:

|  |
| --- |
| $ **python ip-route.py**  Number of routes per interface  --------------------------------------------  {'0/0/0/0': 14, '0/0/0/1': 6, '0/0/0/2': 20} |

## Tabulate OS Types

In this exercise, you will use **if** and **elif** statements to

and tabulate information about a set of devices. Use the **~/Desktop/PRNE/section09/devices** file as the input for your application.

**Step 7**

Create a dictionary of [OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS) types, with the keys "Cisco IOS", "Cisco Nexus", "Cisco IOS-XR", and "Cisco IOS-XE".

### **Answer**

|  |
| --- |
| from pprint import pprint  # Print heading  print ''  print 'Counts of different OS-types for all devices'  print '============================================'  os\_types = { 'Cisco IOS': {'count':0, 'devs':[] },  'Cisco Nexus': {'count':0, 'devs':[] },  'Cisco IOS-XR': {'count':0, 'devs':[] },  'Cisco IOS-XE': {'count':0, 'devs':[] } } |

**Step 8**

Read in the file of devices.

### **Answer**

|  |
| --- |
| # Read all lines of device information from file  file = open('devices','r')  for line in file:  device\_info\_list = line.strip().split(',') # Get device info into list |

**Note**

Make sure you are in the **~/Desktop/PRNE/section09/** directory before running your script. This is because the **devices** file the script is attempting to open is in current working directory.

**Step 9**

For every OS type, create a dictionary with two items: a count of the number of devices of this OS type, and a list of device names of the devices of this device type. Also, for every device, determine the OS type.

### **Answer**

|  |
| --- |
| # Put device information into dictionary for this one device  device\_info = {} # Create a dictionary of device info  device\_info['name'] = device\_info\_list[0]  device\_info['os-type'] = device\_info\_list[1]  name = device\_info['name'] # get the device name  os = device\_info['os-type'] # get the OS-type for comparisons |

**Step 10**

Depending on the OS type, increment the count of the correct OS type in your dictionary, and add the name of the device to the list of devices.

### **Answer**

|  |
| --- |
| # Based on the OS-type, increment the count and add name to list  if os == 'ios':  os\_types['Cisco IOS']['count'] += 1  os\_types['Cisco IOS']['devs'].append(name)  elif os == 'nx-os':  os\_types['Cisco Nexus']['count'] += 1  os\_types['Cisco Nexus']['devs'].append(name)  elif os == 'ios-xr':  os\_types['Cisco IOS-XR']['count'] += 1  os\_types['Cisco IOS-XR']['devs'].append(name)  elif os == 'ios-xe':  os\_types['Cisco IOS-XE']['count'] += 1  os\_types['Cisco IOS-XE']['devs'].append(name)  else:  print " Warning: unknown device type:", os |

**Step 11**

When all lines have been read print the results.

### **Answer**

|  |
| --- |
| pprint(os\_types)  print '' # Print final blank line  file.close() # Close the file since we are done with it |

**Step 12**

Run your application and verify the output.

### **Answer**

Your output should look similar to:

|  |
| --- |
| $ **python os-types.py**  Counts of different OS-types for all devices  ============================================  {'Cisco IOS': {'count': 2, 'devs': ['d01-is'. 'd02-is']},  'Cisco IOS-XE': {'count': 2, 'devs': ['d07-xe'. 'd08-xe']},  'Cisco IOS-XR': {'count': 2, 'devs': ['d05-xr'. 'd06-xre']},  'Cisco Nexus': {'count': 2, 'devs': ['d03-nx'. 'd04-nx']}, |

## Tabulate OSPF Interfaces

In this exercise, you will read route info from a device. You will tabulate and print routes per interface.

**Step 13**

Connect to r1.

### **Answer**

In addition to connecting to r1, the code block shows the regular expressions used for matching.

|  |
| --- |
| import pexpect  from pprint import pprint  import re  # Print heading  print ''  print 'Interfaces, routes list, routes details'  print '---------------------------------------'  # Create regular expressions to match interfaces and OSPF  OSPF\_pattern = re.compile('^O')  intf\_pattern = re.compile('(GigabitEthernet)([0-9]\/[0-9])')  # Create regular expressions to match prefix and routes  prefix\_pattern = re.compile('^O.{8}([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\/[0-9]{1,2})')  route\_pattern = re.compile('via ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})')  # Connect to device and run 'show ip route' command  print '--- connecting telnet 10.30.30.1 with cisco/cisco'  session = pexpect.spawn('telnet 10.30.30.1', timeout=20)  result = session.expect(['Username:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print 'Timeout or unexpected reply from device'  exit()  # Successfully got username prompt, enter username  session.sendline('cisco')  result = session.expect('Password:')  # Enter password  session.sendline('cisco')  result = session.expect('>') |

**Step 14**

Read the route information using **show ip route.**

### **Answer**

|  |
| --- |
| # Must set terminal length to zero for long replies  print '--- setting terminal length to 0'  session.sendline('terminal length 0')  result = session.expect('>')  # Run the 'show ip route' commanmd on device  print '--- successfully logged into device, performing show ip route command'  session.sendline('show ip route')  result = session.expect('>')  # Print out the output of the command, for comparison  print '--- show ip route output:'  show\_ip\_route\_output = session.before  print show\_ip\_route\_output  # Get the output from the command into a list of lines from the output  routes\_list = show\_ip\_route\_output.splitlines() |

**Step 15**

Create a list of [OSPF](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OSPF) routes using the output of the **show ip route** command.

### **Answer**

|  |
| --- |
| intf\_routes= {} # Create dictionary to hold number of routes per interface  # Go through the list of routes to get routes per interface  for route in routes\_list:  OSPF\_match = OSPF\_pattern.search(route)  if OSPF\_match:  intf\_match = intf\_pattern.search( route ) # Match for Gigabit Ethernet  # Check to see if we matched the Gig Ethernet string  if intf\_match:  intf = intf\_match.group(2) # get the interface from the match |

**Step 16**

For every route created above, create a dictionary holding the destination [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address)/subnet, and the next hop.

### **Answer**

|  |
| --- |
| if intf not in intf\_routes: # If route list not yet created, do so now  intf\_routes[intf] = []  # Extract the prefix (destination IP address/subnet)  prefix\_match = prefix\_pattern.search(route)  prefix = prefix\_match.group(1)  # Extract the route  route\_match = route\_pattern.search(route)  next\_hop = route\_match.group(1)  # Create dictionary for this route, and add it to the list  route = {'prefix':prefix,'next-hop':next\_hop}  intf\_routes[intf].append(route) |

**Step 17**

Print the data that you have tabulated.

### **Answer**

|  |
| --- |
| pprint(intf\_routes)  print '' # Print final blank line |

**Step 18**

Run your application and verify your output.

### **Answer**

Your output should look similar to:

|  |
| --- |
| $ **python challenge.py**  Interfaces, routes list, routes details  ---------------------------------------  --- connecting telnet 10.30.30.1 with cisco/cisco  --- setting terminal length to 0  --- successfully logged into device, performing show ip route command  --- show ip route output:  show ip route  Codes: L - local, C - connected, S - static, R - RIP, M - mobile, B - BGP  D - EIGRP, EX - EIGRP external, O - OSPF, IA - OSPF inter area  N1 - OSPF NSSA external type 1, N2 - OSPF NSSA external type 2  E1 - OSPF external type 1, E2 - OSPF external type 2  i - IS-IS, su - IS-IS summary, L1 - IS-IS level-1, L2 - IS-IS level-2,  ia - IS-IS inter area, \* - candidate default, U - per-user static root  o - ODR, P - periodic downloaded static route, H - NHRP, l - LISP  a - application route  + - replicated root, % - next hop override  Gateway of last resort not set  10.0.0.0/8 is variably subnetted, 11 subnets, 3 masks  C 10.1.1.0/24 is directly connected, GigabitEthernet0/1  L 10.1.1.1/32 is directly connected, GigabitEthernet0/1  C 10.1.2.0/30 is directly connected, GigabitEthernet0/2  L 10.1.2.1/32 is directly connected, GigabitEthernet0/2  C 10.1.3.0/30 is directly connected, GigabitEthernet0/3  L 10.1.3.1/32 is directly connected, GigabitEthernet0/3  O 10.2.3.0/30 [110/2] via 10.1.3.2, 00:04:14, GigabitEthernet0/3  [110/2] via 10.1.2.2, 00:04:14, GigabitEthernet0/2  O E2 10.11.12.0/24 [110/20] via 10.1.3.2, 00:04:14, GigabitEthernet0/3  [110/20] via 10.1.2.2, 00:04:24, GigabitEthernet0/2  C 10.30.30.1/32 is directly connected, Loopback0  O E2 10.30.30.2/32 [110/20] via 10.1.2.2, 00:04:24, GigabitEthernet0/2  O E2 10.30.30.3/32 [110/20] via 10.1.3.2, 00:04:14, GigabitEthernet0/3  r1  {'0/2': [{'next-hop': '10.1.2.2', 'prefix': '10.30.30.2/32'}],  '0/3': [{'next-hop': '10.1.3.2', 'prefix': '10.2.3.0/30'},  {'next-hop': '10.1.3.2', 'prefix': '10.11.12.0/24'},  {'next-hop': '10.1.3.2', 'prefix': '10.30.30.3/32'}]}  $ |

**Section 10: Looping with for and While**

10.1 Introduction

A primary goal of network programmability is to take a task that a person could do manually, such as upgrading IOS on a switch, and automate the process. For example, you can upgrade 1,000 switches using the same process with no mistakes in a small fraction of the time it would take a person to do it manually.

If you want to perform the same function over and over, the programming mechanism you will use to do that is called a loop. A loop is a way to perform the same set of programming instructions over and over, such as connecting to a switch and issuing an upgrade command, and stop at some point once you have accomplished whatever task you are performing. There are different types of loop statements such as for, while and range statements that are useful in different scenarios.

If you are creating a script to upgrade your top-of-rack switches in a data center, your script may start by opening a file that contains a list of all your switches. Opening a file and process every line in the file is a common place to use a while loop:

file = open('devices','r')

line = file.readline()

while line:

# store the switch name and IP address in a dictionary

# read the next line

line = file.readline()

Once your script has read the list of top-of-rack switches into a dictionary data structure, you could use a “for” loop to process each switch in the dictionary.

for key,value in switch\_dict.items():

switch\_name = key

ip\_addr = value

# connect to switch at ip\_addr and perform an upgrade

If you want to limit the upgrade to the first 10 racks in the data center, you might use a **range**statement instead of a **for** loop to limit processing to a subset of the switches. If you have a switch naming convention of sw-1, sw-2, sw-3, you could accomplish this with the following **range** statement:

for index in range(1,10):

switch\_name = “sw-“ + index

ip\_addr = switch\_dict[switch\_name]

# connect to switch at ip\_addr and perform an upgrade

There are three major types of looping statements: **for**, **while**, and **range**.

These looping mechanisms are used as follows:

* **for loop:** **for** loops are used to iterate through a sequence of items, such as a list, or a dictionary, or through a set of lines in a file.
* **while loop:** **while** loops provide similar functionality as **for** loops, continuing to repeat until a condition is met.
* **range():** In a **for** loop, a range can be used to iterate over a numerical sequence.

10.2 Looping with for and while

**for** loops are used to iterate over items in a sequence. Data structures such as lists and tuples are sequenced which means they can be iterated or stepped through.

The general structure of a **for** loop, in simplified form, can be thought of as follows:

for item in item-list:

# code block to execute for every iteration of the for loop

The process that takes place is as follows:

* The variable **item** is set to reference the first value in the **item\_list**.
* The code block is executed.
* The program returns to the top of the **for** loop, where **item** is set to reference the next value in the sequence.
* The code block is executed, control returns to the top of the **for** loop, where **item** is set to the next value.
* The process continues until the **for** loop reaches the last **item** in **item\_list**.

Some examples of simple **for** loops:

for line in file:

The example is opening a file and reading it line-by-line. The **for** statement starts with the first line in the file, executes the code block, then returns and executes the code block for the second line in the file, then the third, and so on.

The first time through, **line** references the first line in the file; the second time through, it references the second line, and so on.

The next example takes a list of devices, and one by one, iterates through the list. The first time through, **device** references the first item in the list of devices, the second time through it references the second item in the list, and so on.

for device in devices\_list:

## Example for Loop: Reading Lines

Python provides several built-in objects that can be iterated across, using very simple syntax. One example is reading the lines of a text file.

The process of reading lines from a file in Python because certain files are iterable. For a text file, the process consists of:

* Opening the file
* Iterating through the file line-by-line using a **for** statement

The following example shows this process:

file = open('devices','r')

for line in file: # iterate over all lines in file

device\_info\_list = line.split(',')

The **for** statement above causes the code to loop through the process of reading a file one line at a time, taking the specified action in the code block below the **for** statement.

The **for** statement effectively says the following:

* Start with the file 'file'
* Take the first line and assign it to 'line'
* Perform the statements in the code block (in this case, taking the text line, splitting it into individual items based on commas (','), and creating a list from those items)
* Repeat until all lines of the file have been processed.

Other languages require some type of index for iteration, and require explicitly reading a line of the file.

10.2 for loops

**for** loops are used to iterate over items in a sequence. Data structures such as lists and tuples are sequenced which means they can be iterated or stepped through.

The general structure of a **for** loop, in simplified form, can be thought of as follows:

for item in item-list:

# code block to execute for every iteration of the for loop

The process that takes place is as follows:

* The variable **item** is set to reference the first value in the **item\_list**.
* The code block is executed.
* The program returns to the top of the **for** loop, where **item** is set to reference the next value in the sequence.
* The code block is executed, control returns to the top of the **for** loop, where **item** is set to the next value.
* The process continues until the **for** loop reaches the last **item** in **item\_list**.

Some examples of simple **for** loops:

for line in file:

The example is opening a file and reading it line-by-line. The **for** statement starts with the first line in the file, executes the code block, then returns and executes the code block for the second line in the file, then the third, and so on.

The first time through, **line** references the first line in the file; the second time through, it references the second line, and so on.

The next example takes a list of devices, and one by one, iterates through the list. The first time through, **device** references the first item in the list of devices, the second time through it references the second item in the list, and so on.

for device in devices\_list:

**Example for Loop: Reading Lines**

Python provides several built-in objects that can be iterated across, using very simple syntax. One example is reading the lines of a text file.

The process of reading lines from a file in Python because certain files are iterable. For a text file, the process consists of:

* Opening the file
* Iterating through the file line-by-line using a **for** statement

The following example shows this process:

file = open('devices','r')

for line in file: # iterate over all lines in file

device\_info\_list = line.split(',')

The **for** statement above causes the code to loop through the process of reading a file one line at a time, taking the specified action in the code block below the **for** statement.

The **for** statement effectively says the following:

* Start with the file 'file'
* Take the first line and assign it to 'line'
* Perform the statements in the code block (in this case, taking the text line, splitting it into individual items based on commas (','), and creating a list from those items)
* Repeat until all lines of the file have been processed.

Other languages require some type of index for iteration, and require explicitly reading a line of the file.

10.3 For Loops with Data Structures

In the same way that text files are iterable, data structures such as lists, tuples, and dictionaries are iterable as well.

## For Loop: Items from a List

The process of iterating through a list is as simple as reading a line from a file. Assuming you have a Python list of devices that is called devices\_list, the code is as follows:

for device in devices\_list: # iterate over list

# do something with 'device'

Python will start with the first item in the list (remember that lists are sequences, so it will be in that order), and in the example above, device is assigned to reference the first item in the devices\_list. The code block below is executed, then control returns back to the top, with device now referencing the second item in the list.

## For Loop: Items from a Dictionary

With Python, you can also iterate through items in a dictionary, but the syntax is a bit different and may be somewhat confusing at first.

Rather than having just one variable which gets assigned to reference the item in the iterated sequence, the for loop for dictionary items returns two references, one for the key, and one for the value:

for key,value in device\_dict.items():

ip\_addr = key

device\_info = value

In the example above, both the key and the value are retrieved from the dictionary, using the items() method. With both the key and value, your application can take the appropriate action, in this case setting the [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms) and device info variables.

## For Loop: Keys from a Dictionary

With Python, you can also iterate through just the keys in a dictionary; the syntax is a bit more like iterating through a list.

The **for** loop returns keys from the dictionary, and your code can make use of the keys in whatever manner is appropriate:

for key in device\_dict:

ip\_addr = key

device\_info = device\_dict[ip\_addr]

In the example above, the **for** loop returns the key from the device dictionary, and that key is then used to access the value in the dictionary. This example performs the same function as the previous example, but does it by iterating through the keys only.

10.4 Using range

There are times when you wish to iterate through a sequence of numbers, rather than through a sequence of lines in a file, or items in a data structure.

Using **range** allows you to iterate across a sequence of numerical values, specifying a starting value, a stopping value, and a 'step' value.

for num in range(start, stop, step):

The variable **num** will initially be set to the start value, and every iteration **num** will be incremented by the step value. Iteration will stop when **num** reaches the stop value. You can omit the start value (default is 0), and the step value (default is 1). The stop value must be specified.

Using range can be especially helpful when utilizing indices for lists, or if you to do something for a specific number of times.

10.5 while loop

The **while** loop is useful for repeating code over and over again, as long as a certain condition is met.

The **while** statement specifies the condition to be evaluated every time through the loop. As long as the condition is met, the looping continues.

while condition:

# code block for the while loop

Note that **while** loops do not have automatic iterators, meaning that if you are wishing to go through a sequence of items of some sort, the indexing and selection of items for each iteration must be done manually.

One of the more common while statements that you will see is a **while true** loop.

while True:

# code block for the while loop,

# including checking for some event that breaks out of loop

In this type of **while** loop, the program will repeat the code block over and over, until some event occurs. Some examples of this type of **while** loop are:

* Looping until the user decides to terminate the program by entering appropriate quitting value.
* Looping, reading other types of [I/O](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=I%2FO&type=terms) such as messages or events, until the program is externally terminated

Overall, **while** loops have certain uses as described above, but **for** loops tend to be more popular.

## while Loop: Read File Example

The following example shows what reading a file line-by-line would look like, done with a **while** loop.

The code for reading devices from a file is as follows:

file = open('devices','r')

line = file.readline()

while line:

# do something

line = file.readline()

## while Loop: Input Example

The following example shows an example of having a 'while true' loop to repeat forever, until the user decides to terminate the program.

The code for reading input from the user is as follows:

while True:

name = raw\_input('Enter device name or <cr>:')

print 'Name:',name if name else exit()

There are a few things to note in the above example:

* 'while true': the code is looping forever using **while true**, at least until some event causes the code to terminate the loop.
* The example is using an input function that is called **raw\_input**, which prints the text that is provided, and awaits input from the user, followed by a carriage return.
* The code quits the **while** loop when the user presses carriage return without entering any data.
* The last statement is an inline **if** statement, which reads as follows: "Print the value of 'name', if the value of 'name' is not empty; if it is empty, then exit()"

10.6 Using Loops

# Using Loops

Iterating through a list of devices is a common requirement for network programmability. In the exercises, you will use **for** loops and **while** loops to read information from a file and to iterate through a list of devices.

## for Loop

In this exercise you will:

* Use a **for** loop to read lines from a file
* Use a **for** loop to iterate through devices in a list
* Use Python formatting capabilities to print nice output

**Step 1**

This exercise will use the **devices** file that is located in the **~/Desktop/PRNE/section10** folder. Note that two of the devices have the same [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms)—this is intentional:

|  |
| --- |
| d01-is,ios,Mgmt:10.3.21.5,Version 5.3.1,cisco,cisco  d02-is,ios,Mgmt:10.3.21.6,Version 4.22.18,cisco,cisco  d03-nx,nx-os,Mgmt:10.3.21.7,Version 5.3.1,cisco,cisco  d04-nx,nx-os,Mgmt:10.3.21.8,Version 5.3.1,cisco,cisco  d05-xr,ios-xr,Mgmt:10.3.21.8,Version 4.16.9,cisco,cisco  d06-xr,ios-xr,Mgmt:10.3.21.10,Version 5.3.0,cisco,cisco  d07-xe,ios-xe,Mgmt:10.3.21.19,Version 4.16.0,cisco,cisco  d08-xe,ios-xe,Mgmt:10.3.21.22,Version 5.3.0,cisco,cisco |

**Step 2**

Create a Python application that uses a **for** loop to iterate through lines of the **devices** file. You will read information about all devices from the file, one line at a time, placing the devices into a list. For each device, store the device information in a list. The result of reading this information should be a list of devices, where each device is a list of device information.

### **Answer**

|  |
| --- |
| devices\_list = [] # Create thprinte outer list for all devices  # Read in the devices from the file  file = open('devices','r')  for line in file:  device\_info\_list = line.strip().split(',') # Get device info into list  devices\_list.append(device\_info\_list)  file.close() # Close the file since we are done with it |

**Step 3**

Create a second **for** loop that iterates through the list of devices. For every device, print the device information. The output should be a nice table, such as the following:

|  |
| --- |
| Name OS-type IP address Software  ------- -------- ------------------- --------------------  d01-is ios Mgmt:10.3.21.5 Version 5.3.1  d02-is ios Mgmt:10.3.21.6 Version 4.22.18  d03-nx nx-os Mgmt:10.3.21.7 Version 5.3.1  d04-nx nx-os Mgmt:10.3.21.8 Version 5.3.1  d05-xr ios-xr Mgmt:10.3.21.8 Version 4.16.9  d06-xr ios-xr Mgmt:10.3.21.10 Version 5.3.0  d07-xe ios-xe Mgmt:10.3.21.19 Version 4.16.0  d08-xe ios-xe Mgmt:10.3.21.22 Version 5.3.0 |

You will need to use print formatting functionality in Python. The simplest version of the print statement will look similar to:

|  |
| --- |
| print '{0:8} {1:8} {2:20} {3:20}'.format(device[0],device[1],  device[2],device[3] |

### **Answer**

|  |
| --- |
| print ''  print 'Name OS-type IP address Software '  print '------ ------- ------------------ ------------------'  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:8} {1:8} {2:20} {3:20}'.format(device[0],device[1],  device[2],device[3])  print '' |

## while Loop

In this exercise you will:

* Use a **while** loop to read input from a file, utilizing the **readline()** function
* Use a **while** loop to iterate through devices in a list, using your own index variable in order to manually perform the iteration.

This exercise will also use the devices file in the **~/Desktop/PRNE/section10** folder.

**Step 4**

Create a Python application that uses a **while** loop to iterate through lines of the **devices** file. You will read information about all devices from the file, one line at a time, placing the devices into a list. For each device, store the device information in a dictionary. The result of reading this information should be a list of devices where every device is a dictionary of device information.

Note that reading information from a text file requires reading the lines manually using **file.readline()**.

### **Answer**

NOTE: You must read one line of the file before executing the **while** loop. If you do not, the condition will be false and the code block associated with the **while** loop will not be executed.

|  |
| --- |
| devices\_list = [] # Create the outer list for all devices  file = open('devices','r')  line = file.readline()  while line:  device\_info\_list = line.strip().split(',') # Get device info into list  # Put device information into dictionary for this one device  device\_info = {} # Create the inner dictionary of device info  device\_info['name'] = device\_info\_list[0]  device\_info['os-type'] = device\_info\_list[1]  device\_info['ip'] = device\_info\_list[2]  device\_info['version'] = device\_info\_list[3]  # Now append our device and its info onto our 'devices' list  devices\_list.append(device\_info)  line = file.readline()  file.close() # Close the file since we are done with it |

**Step 5**

Create a second **while** loop that iterates through the list of devices. For every device, print the device information.

The output should be a nice table, such as the following:

|  |
| --- |
| Name OS-type IP address Software  ------- -------- ------------------- --------------------  d01-is ios Mgmt:10.3.21.5 Version 5.3.1  d02-is ios Mgmt:10.3.21.6 Version 4.22.18  d03-nx nx-os Mgmt:10.3.21.7 Version 5.3.1  d04-nx nx-os Mgmt:10.3.21.8 Version 5.3.1  d05-xr ios-xr Mgmt:10.3.21.8 Version 4.16.9  d06-xr ios-xr Mgmt:10.3.21.10 Version 5.3.0  d07-xe ios-xe Mgmt:10.3.21.19 Version 4.16.0  d08-xe ios-xe Mgmt:10.3.21.22 Version 5.3.0 |

### **Answer**

Important: you will have to manually iterate through the indexes of the list of devices. Setting the list index to 0 at the start, check that the index is less than the length of the list as the while statement condition, and increment the index at the bottom of the while loop.

|  |
| --- |
| # Use while loop to print the results  print ''  print 'Name OS-type IP address Software '  print '------ ------- ------------------ ------------------'  index = 0  while index < len(devices\_list):  device = devices\_list[index]  print '{0:8} {1:8} {2:20} {3:20}'.format(device['name'],  device['os-type'],  device['ip'],  device['version'])  index += 1  print '' |

10.7 Looping Control

You may wish to either exit your loop, or continue on to the next iteration without continuing to process the code block. These actions are possible with the **break** and **continue** statements

The **break** statement tells Python that you wish to exit the loop immediately. Control then proceeds at the first line following the code block for your **for** or **while** statement.

The **continue** statement tells Python that you wish to proceed to the next iteration of the loop immediately. Control then proceeds at the top of the code block for the loop. For a **for** loop, iteration takes place as it normally would, which means that the iteration item (for example, line in a file, or item in a list) is set to the next value.

## Break Example: while Loop

The following code shows an example of breaking out of the middle of a **while** loop.

while True:

value = input("Enter value, 'exit' to exit")

if value == 'exit':

break

# code to handle user input

print "User entered 'exit'"

In the example, the loop will continue until the user types in 'exit'. When this occurs, **value** with match the exit string, and **break** is called, causing control to pass to the line after the end of the code block.

## Break Example: for Loop

Breaking out of a **for** loop works the same as **while** loops.

The following code shows an example of breaking out in the middle of a **for** loop.

for device in devices\_list:

if device.ip == search\_ip\_addr:

print 'Found device:', device

break

# do other stuff

# rest of the program

In the example, the devices list is examined, one device at a time, looking for a device with an [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms) that matches the given search\_ip\_address. Once a match is found, it is printed, and the loop breaks, continuing at the line of code immediately following the **for** loop code block.

## Continue Example: for Loop

The **continue** statement causes Python to go back to the top of the loop, and execute the code block again. If the looping construct is a **for** loop, then iteration will take place, and the next item in the iteration will be used as the code block is executed.

The following example shows a **for** loop with a **continue** statement. The code iterates across all devices in the device list, until a match is found. In this case, however, the code returns to the next iteration.

for device in devices\_list:

if device.ip == search\_ip\_addr:

print 'Found device:', device

continue

# do other stuff with non-matching device

The idea in the code sequence above is that there may be more than one matching device, so this code finds them all.

10.8 Nested Loops

Sometimes it will be necessary to implement code that has multiple looping blocks nested within one another. In these situations, the inner loop will execute until it exits for some reason, continuing with the next iteration of the outer loop.

The following example shows an outer **while** loop, which continues to loop until some event causes the loop to exit (for example, user input). The details of exiting are not shown in this example for simplification purposes.

The inner loop iterates through all devices in a list of devices, looking for a specific device name. If the device name is found, the inner loop is exited, and iterations continue for the outer while statement.

while True:

name = input("Enter name:")

for device in devices\_list:

if name == device.name:

print 'Found device:', name

break

else:

print 'Did not find device:', name

Note: the inner **for** statement iterates over devices in the file; when a match is found, it is printed, and the **for** loop is exited. But look carefully at the **else** clause. It is important to understand where the **else** belongs. Looking closely at the indentation, you can see that the else belongs with the **for** statement.

10.9 else Command in for or while loops

When a loop exits, it is sometimes important to know whether the loop ended because the iteration was complete, or whether the loop exited as a result of a **break** statement. With other languages, you can set a flag to let you know if exiting was as a result of finding something or not; with Python, there can be an optional **else** clause at the end of the **for** or **while** loop, specifically for this purpose.

The code shows the structure of an **else** statement associated with a **for** loop.

for item in item-list:

# code for the for loop

if condition:

break

# more code for the for loop

else:

# code block for when loop completes normally

The indentation of the **else** shows that it is a clause at the end of the **for** loop. The behavior is:

* If the **for** loop ran to completion, the **else** code block is executed.
* If the **for** loop exited for some reason via a **break** statement, the **else** code block is not executed.

The following example shows a use of this **else** clause:

for device in devices\_list:

if name == device.name:

print 'Found device:', name

break

else:

print 'Did not find device:', name

In the example, the code is iterating across devices in a list of devices, looking for a device with the specified name. If the **for**loop goes through all devices in the list, it will execute the **else** clause, printing an indication that no device with that name was found. If the device with the specified name is found, the **break** statement will be executed and the **else** clause is not executed.

The code structure and behavior for a **while-else** is the same as with the **for** loop.

10.10 Lab Using break and continue

## Highlight Duplicate Device Information

In this exercise, you will read the information from the devices file in the PRNE/section10 folder and highlight any duplicate[IP addresses](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms) found. In order to write this application, you will:

* Use a **for** loop to read in device information
* Use a set to store known IP addresses
* Do a comparison of each device IP address against a set of known IP addresses.
* Use a continue statement to move to the next iteration if a match is found.

**Step 1**

Read device information from the file and store in a list.

### **Answer**

|  |
| --- |
| devices\_list = [] # Create the outer list for all devices  # Read in the devices from the file  file = open('devices','r')  for line in file:  device\_info\_list = line.split(',') # Get device info into list  devices\_list.append(device\_info\_list)  file.close() # Close the file since we are done with it |

**Step 2**

Print the device information in a table. If an IP address is a duplicate of one seen before, print a message indicating it is a duplicate.

Your output should look similar to:

|  |
| --- |
| Name OS-type IP address Software  ------- --------- ------------------- --------------------  d01-is ios Mgmt:10.3.21.5 Version 5.3.1  d02-is ios Mgmt:10.3.21.6 Version 4.22.18  d03-nx nx-os Mgmt:10.3.21.7 Version 5.3.1  d04-nx nx-os Mgmt:10.3.21.8 Version 5.3.1  d05-xr ios-xr Mgmt:10.3.21.8 Version 4.16.9 \*Duplicate IP\*  d06-xr ios-xr Mgmt:10.3.21.10 Version 5.3.0  d07-xe ios-xe Mgmt:10.3.21.19 Version 4.16.0  d08-xe ios-xe Mgmt:10.3.21.22 Version 5.3.0 |

### **Answer**

|  |
| --- |
| print ''  print 'Name OS-type IP address Software '  print '------- --------- ------------------- --------------------'  ip\_addresses = set()  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:8} {1:10} {2:20} {3:20}'.format(device[0],device[1],device[2],device[3]),  # Print 'duplicate' if IP address exists for another device  if device[2] in ip\_addresses:  print '\*Duplicate IP!\*'  continue  ip\_addresses.add(device[2])  print ''  print '' |

## Select Devices by IP Address

In this exercise, you will write an application that allows a user to search for device information by entering the IP address. If the IP address is found, the application will display the device information. The application will iterate until the user enters Ctrl-C. In order to write this application, you will:

* Use a **for** loop to read in device information
* Use a **while** loop to allow users to input IP addresses to search for
* Use **range** to iterate through devices in the list
* Use **continue** to continue searching the list
* Use **break** to exit when user presses Ctrl-C
* Use **for...else** to print if the list was exhausted without finding a match

Here are some Python tools that you will find helpful to complete this lab:

* **Input:** Because the Python 2.7, **input** function expects a Python expression as input, and IP addresses are not Python expression, use **raw\_input**:

ip\_address = raw\_input('Enter device IP address to find (Ctrl-C to exit):')

* To capture the Ctrl-C, surround your input code with a **try-except** routine, catching the exception called **KeyboardInterrupt**:
* try:
* ip\_address = raw\_input('Enter device IP address to find (Ctrl-C to exit):')
* except KeyboardInterrupt:

break;

* Recall that the IP address format for these current labs follows the format 'Mgmt:<ip-address>'. To specify a subset of a string, specify the start and end locations. The following statement is from the solution to this lab, specifying that you are looking at **device\_info[2**], which is the second item, or the IP address. The **[5:]** specifies that you are looking at the substring starting at the fifth character, until the end.

if device\_info[2][5:] == ip\_address: # Check to see if device IP is a match

**Step 3**

Read the devices file, creating a list of devices, with each device holding a list of device information. Print a table of the information as you read it in.

### **Answer**

|  |
| --- |
| devices\_list = [] # Create the outer list for all devices  print ''  print 'Idx Name OS-type IP address Software '  print '--- -------- -------- -------------------- --------------------'  index = 0  # Read in the devices from the file  file = open('devices','r')  for line in file:  device\_info = line.split(',') # Get device info into list  devices\_list.append(device\_info)  print '{0:2}: {1:8} {2:8} {3:20} {4:20}'.format(index+1,  device\_info[0],device\_info[1],  device\_info[2],device\_info[3])  index += 1 # increment our index  file.close() # Close the file since we are done with it  print '' |

**Step 4**

Loop forever (while true) reading user input, and matching their input IP address with an item from the list of devices. If the user enters Ctrl-C, exit the program. If the user enters anything else, use it to search for a matching IP address from the list of devices. Use a for statement with a range. The range will go from 0 to the length of the list. If the IP address is found, print a message with the device data. If the IP address is not found, print a message stating: "--- Given IP address not found ---".

### **Answer**

|  |
| --- |
| while True: # Loop forever, until user terminates program  # Request user to input the IP address we will search for  try:  ip\_address = raw\_input('Enter device IP address to find (Ctrl-C to exit):')  except KeyboardInterrupt:  break;  # Loop through our devices looking for a match on IP address  for index in range(0, len(devices\_list)):  device\_info = devices\_list[index] # Get information for this device in the list  if device\_info[2][5:] == ip\_address: # Check to see if device IP is a match  # If a match, print results and stop looking  print '{0:2}: {1:8} {2:8} {3:20} {4:20}'.format(index+1,  device\_info[0],device\_info[1],  device\_info[2],device\_info[3])  break  else:  continue  else: # We get here if we exhausted the device list, IP not found  print '--- Given IP address not found ---'  print '\n'  print 'Device search terminated.\n' |

10.11 Lab Tabulate and print routes per interface

## Learning Lab Procedure

**Step 1**

Connect to device R1 in the lab environment and run the **show ip route** command.

### **Answer**

|  |
| --- |
| routes\_list = [] # Create the list of routes  #-----------------------------------------------------------  # The following code connects to a device and dumps all  # routing information  print '--- connecting telnet 10.30.30.1 with cisco/cisco'  session = pexpect.spawn('telnet 10.30.30.1', timeout=20)  result = session.expect(['Username:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print 'Timout or unexpected reply from device'  exit()  # Successfully got password prompt, logging in with username  session.sendline('cisco')  result = session.expect('Password:')  # Successfully got password prompt, logging in with username  session.sendline('cisco')  result = session.expect('>')  # Must set terminal length to zero for long replies  print '--- setting terminal length to 0'  session.sendline('terminal length 0')  result = session.expect('>')  # Execute the 'show ip route' command to get routing info  print '--- executing: show ip route'  session.sendline('show ip route')  result = session.expect('>')  # Get output from ip route command  print '--- getting ip route command output'  show\_ip\_route\_output = session.before  print ''  print 'IP route output'  print '----------------------------------------------------'  print show\_ip\_route\_output  print '----------------------------------------------------'  print '' |

**Step 2**

Using the output of the above show ip route command, create a list of destination IP address prefixes. For each destination, you will create a list of potential next hops, including next hop IP address, interface, and any other information you choose.

### **Answer**

Note: It may be more efficient to use a dictionary here in a real-world situation, but since this module focuses on looping a list was used to show iterating through the list of routes.

|  |
| --- |
| # Get routing information into list  routes\_list = show\_ip\_route\_output.splitlines() |

**Step 3**

Allow the user to input a destination IP address prefix.

### **Answer**

|  |
| --- |
| while True: # Loop forever, until user terminates program  # Request user to input the IP destination route prefix we will search for  try:  ip\_address = raw\_input('Enter IP destination address to find (Ctrl-C to exit):')  except KeyboardInterrupt:  break; |

**Step 4**

Go through the list of destination IP prefixes, searching for a match learned via OSPF. Print the route information if there is a match. If there is no match, print "--- Given route prefix not found ---'.

### **Answer**

|  |
| --- |
| # Set the pattern for matching OSPF routes  route\_pattern = re.compile('^O.{8}([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})')  # Loop through our devices looking for a match on IP address  for route in routes\_list:  # Search for our route string, and continue to next iteration if not found  route\_match = route\_pattern.search(route)  if not route\_match: continue  # Found our IP address, print out route information  if route\_match.group(1) == ip\_address:  route\_info = route.split(',')  print ' ---- Route: ', route\_info[0][5:].strip()  print ' ---- Time: ', route\_info[1].strip()  print ' ---- Interface: ', route\_info[2].strip()  print ''  break  else: # We get here if we exhausted the device list, IP not found  print '--- Given route prefix not found ---'  print '\n'  print 'Route search terminated.\n'  session.sendline('quit')  session.kill(0) |

Chapter 11 Function

11.1 Introduction

One of the most important facets of any programming language, including Python, is the ability to group certain pieces of code into functions, sometimes called methods, or subroutines in other languages.

Very long sequences of code become complicated, difficult to write, read, and debug. By taking certain parts of the code and organizing them into functions, each with a specific purpose, you improve your application during development time, and for readability once your code gets passed along to another set of developers.

Suitable blocks of functionality in a networking environment are:

* Connecting to a device
* Executing a command on a device
* Printing device information
* Searching for specific data from a device

Another advantage of using functions is to reduce, and hopefully eliminate, replicated code. If there is a specific operation that is invoked over and over in your application, it is a bad idea to repeat that code everywhere that it is needed.

A far better choice is to put that repeated code into a single function, which can be called from multiple places in your code.

* **Function Parameters:** What happens if the same code needs to be called, but with minor modifications to certain details? This is where the ability to pass arguments to your function is important. Using this mechanism, you can call the same code, but with varying input parameters, and have the code be reused for each instance, without code replication.
* **Function Return Values:** What happens if the code needs to create some resulting value that is needed by the calling code? This is where the ability to return values to the calling program is useful; values can be as simple as a number, a string, or a large data structure.

This lesson will examine:

* **Definition:** How to define a function.
* **Call:** How to call a function.
* **Scope:** How variable names work within a function
* **Documentation:** How to document your function.

11.2 Function Definition Example

This example defines a function that is called 'connect', which connects to the given device using Pexpect and returns the resulting 'session' object to the caller.

#-----------------------------------------------------------

# The following code connects to a device

def connect(dev\_ip,username,password):

print '--- attempting to: ssh '+username+'@'+dev\_ip

session = pexpect.spawn('ssh '+username+'@'+dev\_ip, timeout=20)

result = session.expect(['password:', pexpect.TIMEOUT])

# Check for failure

if result != 0:

print '--- Timout or unexpected reply from device'

return 0

print '--- attempting to: password: '+password

# Successfully got password prompt, logging in with password

session.sendline(password)

session.expect('#')

return session # return pexpect session object to caller

#-----------------------------------------------------------

Notice the following:

* **Function name:** connect, the name that will be used by the calling code.
* **Function parameters:** the device IP address, username, password, that will be used by the function when attempting to connect to the device.
* **Function body:** all the code to use Pexpect to execute an ssh command using the IP address and username, and to enter in the password when prompted.
* **Function return value:** session, the object returned to the calling code.

## Calling a Function

To complete the picture, here is a sample of the code that is used to call the function:

session = connect('10.0.0.1','cisco','cisco')

The calling code shows the following:

* **Function name:** connect (same as the name in the function definition).
* **Function parameters:** the actual values that are passed are the IP address ('10.0.0.1'), the username ('cisco'), and the password ('cisco').
* **Function return value:** 'session', which is the value returned as an object representing the [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH&type=terms) Pexpect session.

There are a couple of things to note:

* The function name must match exactly the value specified in the function definition.
* The values that are passed to the function as parameters – which are called the 'arguments' – are the actual values that will be used by the function. You can pass specific constant variables, as is done here, or you can pass a variable name which points to the actual values.
* The returned value goes on the left-hand side of an assignment statement – which is how the value is returned to your calling code. The actual name of the variable you use to hold the returned value can be anything; in the example, it just happens to be the same as the returned value specified in the function definition.

## Documenting a Function

Functions are for use by other parts of your code, and perhaps even for use by other people's code. Therefore it is important to make sure that they are documented.

The standard manner for documenting functions in Python is by using a **docstring**, which is a long string literal spanning multiple lines. This **docstring** will document the purpose of your function, the parameters of your function, and the return values of your function.

The following is an example of a **docstring** for the 'connect' function:

dev connect(dev\_ip, username, password)

"""

Connects to device using pexpect

:dev\_ip: The IP address of the device

:username: The username for logging in

:password: The password for logging in

=return value: successful: pexpect session object

not successful: 0

"""

There are different recommendations for how exactly to format your **docstring**, depending on your organization and its preferences. Whatever format you choose, a **docstring** should specify the following items:

* The purpose of the function
* The parameters passed to the function
* The values returned by the function

11.3 Scope

The idea of 'scope' when dealing with software involves the question of visibility. For example:

* If a variable is defined inside a function, it is not visible to code outside the function.
* If a variable is defined outside of a function, it is visible to code inside a function if the variable is global.
* A variable 'global' if it is defined outside of any function.

What this means is that variables that are defined within a function are only visible inside the function. If you want your data to be known outside the function, you should return it as a return value.

If you define a global variable, it is visible to all other code in the same module. However, good programming practice dictates that you try not to overuse global variables, and accessing them within functions is not generally a good idea. A better solution is to pass the necessary values into the function as parameters.

It is important to structure your application in such a way as to make it easy to distinguish the following items:

* **Functions:** Functions should be defined at the beginning of your application or module.
* **Main code:** Main code should be placed at the bottom of your application file.
* **Global variables:** Global variables are accessible from all code. For readability, global variables should either be initially defined (for example, set to an empty list, dictionary, and so on) at the very top of your application file, or at the very top of your main code.

The following is an example of how you may want to structure your application, now that you are using functions and creating applications that are larger and require some level of organization. Application file:

#--- func1 --------------------------------------------------------------

def func1(...) # function definition for func1

#--- func2 --------------------------------------------------------------

def func2(...) # function definition for func2

#--- main ---------------------------------------------------------------

global\_list = [] # global variables outside of function scope

...

x = func1(...)

---

y = func2(...)

Eventually some things will come down to personal preference. But whatever your preference may be, make sure that it is easy to read and understand, for your own benefit, and for the benefit of others who may come after and wish to maintain or make use of your code.

# 11.4Function Parameters

Python provides a rich set of alternatives for passing parameters to a function, including:

* **Positional parameters:** These parameters are in a specific sequence, and the calling code must put their input values (arguments) in the same order as the parameters are listed in the function definition.
* **Keyword parameters:** Python allows parameters to be passed by keyword, meaning that the caller specifies the parameter name, and equals sign, and a value. When done in this way, the parameters can be in any order.
* **Defaults:** Python allows the function to specify default values, so that if the caller does not provide a value for a parameter, the default value is used.
* **Parameter gathering:** Python allows for any number of positional or keyword values to be provided in the function call, and which can be gathered up into a tuple (for positional) or dictionary (for keywords).
* **Parameter passing:** Programming languages had different ways of passing parameters to functions, sometimes referred to as "pass by value" and "pass by reference." Python passes parameters by object reference, since everything is an object.

## Parameters versus Arguments

Terminology:

* When you define the list of items that get passed to your function, those items are called 'parameters'.
* When code calls your function, the items it passes are referred to as 'arguments'.

The following example shows the difference. In the function definition at the top, the items are called parameters. In the actual call to the function at the bottom, the values that are passed to the function are called arguments.

# Define 'connect'

def connect(dev\_IP, username, password):

...

...

...

# Call 'connect'

session = connect('10.0.0.1','cisco','cisco')

## Parameters by Position

Positional parameters are parameters which require calling in a specific order, which is the traditional way that other languages have handled the passing of parameters.

In the example below, on the bottom line of code, the calling program provides the device[IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms), the username, and the password, in the correct order, as specified in the formal definition of the function, which is shown in the top line.

def connect(dev\_IP, username, password):

...

...

...

session = connect('10.0.0.1','cisco','cisco')

## Parameters by Keyword

Keyword parameters allow the calling program to provide the parameters in any order, provided they specify which parameters are which, by using the keywords in the function call.

In the example below, the function definition at the top is identical to when parameters are passed positionally. In fact, the same function definition is used for either method, positional or keyword.

The difference is seen in the function call at the bottom, where the actual keywords (matching the parameters in the definition) are specified in the actual call.

def connect(dev\_IP, username, password):

...

...

...

session = connect(username = 'cisco',

dev\_IP = '10.0.0.1'

password = 'cisco')

## Parameters by Position and Keyword

It is possible to mix positional and keyword parameters, with the rule that all positional parameters come first, in correct order, followed by keyword arguments (which can be in any order).

In the example below, once again the function definition is the same. The calling code however is passing the first argument positionally, without a keyword; and the remaining arguments are being passed by keyword.

def connect(dev\_IP, username, password):

...

...

...

session = connect('10.0.0.1',

password = 'cisco',

username = 'cisco')

Recall that positional parameters must come first, and be in the proper order; and (b) Keyword parameters must come after, and can be in any order.

# 11.5 Default Values

Python allows callers of functions to omit arguments if they so choose.

In such cases, the function definition will have to provide the default values for the parameters that can be omitted. In the example below, the function definition specifies default values for the username and password:

def connect(dev\_IP, username='cisco',

password='cisco'):

...

...

...

session = connect('10.0.0.1')

The calling code in the example specifies only the first argument, and the others are omitted. When called, the function will insert default values for username and password.

# 11.6 Creating Functions

# Creating Functions

In this exercise, you will create functions to read device information from a file, and to print device information in a formatted table.

Lab is currently initializing: 26% complete

## Create a Function Using Global Variables

Create functions without parameters: use a global **devices** list for the purposes of this lab.

**Step 1**

Create a function that will read device information from the devices file located in the PRNE/section11 folder. Store the device information in a Python list.

### **Answer**

|  |
| --- |
| #---- Function to read device information from file -------------------  def read\_device\_info():  # Read in the devices from the file  file = open('devices','r')  for line in file:  device\_info\_list = line.strip().split(',') # Get device info into list  devices\_list.append(device\_info\_list)  file.close() # Close the file since we are done with it |

**Step 2**

Create a function that will take the device information from your list, and print it out in a nicely formatted table.

### **Answer**

|  |
| --- |
| #---- Function to go through devices printing them to table -----------  def print\_device\_info():  print ''  print 'Name OS-type IP address Software '  print '------ ------- ------------------ ------------------'  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:8} {1:8} {2:20} {3:20}'.format(device[0],device[1],  device[2],device[3])  print '' |

**Step 3**

Your 'main' code should simply call your 'read device info' function, then call your 'print device info' function.

### **Answer**

|  |
| --- |
| #---- Main: read device info, then print ------------------------------  devices\_list = [] # Create the outer list for all devices  read\_device\_info()  print\_device\_info() |

## Create a Function Using Parameters

The purpose of this exercise is to exercise passing of a parameter to a function. You will create an application that includes functions for reading a file and printing the data in a table. In this exercise, your application should allow the user to input the name of the file to be read.

**Step 4**

Create a function that will read input from a file. Your function will take the name of the file as a parameter, to be passed by the caller. The function will put the device information into the global devices\_list.

### **Answer**

|  |
| --- |
| #---- Function to read device information from file -------------------  def read\_device\_info(devices\_file):  # Read in the devices from the file  file = open(devices\_file,'r')  for line in file:  device\_info\_list = line.strip().split(',') # Get device info into list  devices\_list.append(device\_info\_list)  file.close() # Close the file since we are done with it |

**Step 5**

Create a function that will print device information from a Python list. The list will be passed as a parameter to your printing function. The information in the list will be the same as what is in the 'devices' file.

### **Answer**

|  |
| --- |
| #---- Function to go through devices printing them to table -----------  def print\_device\_info(list\_of\_devices):  print ''  print 'Name OS-type IP address Software '  print '------ ------- ------------------ ------------------'  # Go through the list of devices, printing out values in nice format  for device in list\_of\_devices:  print '{0:8} {1:8} {2:20} {3:20}'.format(device[0],device[1],  device[2],device[3])  print '' |

**Step 6**

Your main code should create the empty\_devices list, then prompt the user to enter the name of the file containing the device information.

### **Answer**

|  |
| --- |
| #---- Main: read device info, then print ------------------------------  devices\_list = [] # Create empty list of devices  print ''  devices\_file = raw\_input('Enter devices filename:') |

**Step 7**

Once the user has entered the name of the file, your application should call your function to read device info, passing the name of the file as an argument and then call the function to print the device information.

### **Answer**

|  |
| --- |
| read\_device\_info(devices\_file)  print\_device\_info(devices\_list) |

# 11.7 Returning Values

Returning values to callers is one of the more important attributes of functions. In Python, your function can return values to the caller using the 'return' statement.

The returned value is received by the caller using a simple assignment statement:

def connect(dev\_IP, username, password):

...

return session

...

session = connect('10.0.0.1','cisco','cisco')

Your function can return simple values or it can return more complex data such as lists, dictionaries, and so on. It is even possible to return another function.

In the example, the connect function wishes to return the Pexpect 'session' object to the caller, so that more operations can be done with that session. In the function itself, this is done using the return statement, 'return session'.

In the caller, the output of the connect function call is the value of the session, which is assigned to the variable called 'session' in the calling code. The calling code could have assigned the return value to any variable – it could be called 'p\_session' or 'my\_session' or 'connection' or any name you desire.

connection = connect('10.0.0.1','cisco','cisco')

Recall also about a variable's scope. The variable 'session' inside the function itself is entirely separate and distinct from the outer, global variable 'session' that received the return value in the original example.

What values can be returned by a function? In Python there are many options:

It is possible to return single items, as in the following examples:

def connect(...):

return session

...

session = connect(...

In the example above, the function returns Pexpect 'session' object, received from Pexpect after the **connect()** function performed the required connection and login steps.

def get\_ip\_address(...):

return ip\_address

...

ip = get\_ip\_address(...)

In the example above, the function returns a string holding information regarding the [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms) requested by the caller.

def get\_route(...):

return route

...

new\_route = get\_route(...)

In the example above, the get\_route function calculates and returns the route.

It is also possible to return multiple or more complex items:

def get\_devs(...):

return [dev1,dev2,dev3]

...

dev\_list = get\_devs(...)

In the example above, the get\_devs function returns a Python list of devices.

def get\_credentials(...):

return (username, password)

...

cred = get\_credentials(...)

In the example above, the get\_credentials function returns a Python tuple containing the username and password credentials for a specific device.

def get\_credentials(...):

return username, password

...

user,pw = get\_credentials(...)

In the example above, the get\_devs function returns a Python list of devices.

#### Parameters are Passed by Object Reference

For software developers, an important question to ask is "how are the parameters passed to functions?" This is important because if the answer is "passed by value," that means that the function can change the parameter, and it will have no effect on the calling program – that parameter passed by value, from the perspective of the calling program, will not have changed at all.

On the other hand, parameters that are "passed by reference," can actually change; in other words, if the function changes the value of that parameter, then from the calling application, the value has also changed.

Python passes everything by reference, since "everything is an object," which means that, for mutable objects, if you change them in your function, the calling application will see those changes. Specifically, if the parameter passed is mutable such as a list or dictionary, and you append or add an item, that change will be visible to the calling code.

For immutable data types such as strings or integers, what happens is the same thing that happens everywhere in Python – a new object is allocated with the new value. Hence, your changes within your function are not seen by the calling program, because its reference still points at the original piece of data.

#### Gathering Optional Parameters

You may see this type of code in Python applications you inherit, in which the parameters that are listed in the function definition have one or two asterisks ('\*' or '\*\*') preceding them. This indicates that the code is making use of a Python feature allowing the caller to specify a variable number of parameters when calling a function.

Note that this is entirely different from default values for parameters. With default values, Python knows exactly how many parameters are defined for the function; the 'optional' part means that the caller can specify fewer than the total number of parameters, and the ones that are not provided will be assigned default values.

In this situation, the function definition does not know how many parameters will be passed. It could be none, or one, or very many.

Positional:

def function\_name(\*args):

tuple\_input = args

Keyword:

def function\_name(\*\*kwargs):

dict\_input = kwargs

As mentioned, there are two types of optional parameter models, 'positional', and 'keyword', identified by whether the function definition specifies one asterisk (positional), or two asterisks (keyword). The name of the parameter in the function definition does not matter, but by convention, for positional parameters, 'arg' is used, and for keyword parameters, 'kwargs' is used.

For positional parameters passed in this way, Python wraps them up into a tuple, and your function is able to access and use the items in the tuple .

def connect(\*args):

connect\_tuple = args

...

...

session=connect(ip,user,pw)

The example above shows what Python would do if the 'connect' function used a positional optional parameter, rather than three discreet values. In the example, the first item in the tuple would be the device IP address; the second would be the username; and the third would be the password.

For keyword parameters, passed in this way, Python wraps them up into a dictionary (with the keyword part becoming the key, and the actual value becoming the value for the item in the dictionary.

def connect(\*\*kwargs):

connect\_dict = kwargs

...

...

session=connect(ip='10.0.0.1',

user='cisco',

pw='cisco')

The example above shows what Python would do if the 'connect' function used a keyword optional parameter, rather than discreet keyword values. In the example, the connect function would receive a dictionary, with keys being 'ip', 'user', and 'pw', with their accompanying values.

# 11.8 Lab Creating Functions That Return Values

## Create a Connect Function

In this exercise you will:

* Create a function to connect to a device in your virtual network, returning the Pexpect 'session' object.
* Create another function to use the session object to read interface information from the device, returning the output of the **show int brief** command.
* Your main code will call the connect function, call the show interface function, and print the output just as it was received from the device.

**Step 1**

Create a function which takes the [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms), username, and password of the device as input parameters.

The function should use Pexpect to connect to the device. If successful, the function should return the Pexpect 'session' object. If unsuccessful, the function should return 0.

### **Answer**

|  |
| --- |
| import pexpect  #-----------------------------------------------------------  # The following code connects to a device  def connect(dev\_ip,username,password):  """  Connects to device using pexpect  :dev\_ip: The IP address of the device we are connectin to  :username: The username that we should use when logging in  :password: The password that we should use when logging in  =return: pexpect session object if succssful, 0 otherwise  """  print '--- attempting to: telnet ' + dev\_ip  session = pexpect.spawn('telnet ' + dev\_ip, timeout=20)  result = session.expect(['Username:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0  print '--- attempting to: username: ' + username  # Successfully got username prompt, logging with username  session.sendline(username)  result = session.expect(['Password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0  print '--- attempting to: password: ' + password  # Successfully got password prompt, logging in with password  session.sendline(password)  session.expect('>')  return session # return pexpect session object to caller |

**Step 2**

Create a function which takes as input a Pexpect session object, and performs a **show interface brief**command to receive interface information from the device. The function should return the output of the command.

### **Answer**

|  |
| --- |
| #-----------------------------------------------------------  # The following function gets and returns interface information  def show\_int\_summary(session):  """  Runs 'show int summary' command on device and returns  output from device in a string  :session: The pexpect session for communication with device  =return: string of output from device  """  print '--- show interface summary command'  session.sendline('show interface summary')  result = session.expect('>')  print '--- getting interface command output'  show\_int\_brief\_output = session.before  return show\_int\_brief\_output |

**Step 3**

Your main code should call your connect function, passing in the actual IP address, username, and password, and receiving the session object in return. The main code should then call your **show int brief** function, passing in the session object, and receiving the output of the command in return.

### **Answer**

|  |
| --- |
| #------------------------------------------------------------  # Main program: connect to device, show interface, display  if \_\_name\_\_ == '\_\_main\_\_':  session = connect('10.30.30.1','cisco','cisco')  if session == 0:  print '--- Session attempt unsuccessful, exiting.'  exit()  output\_data = show\_int\_summary(session) |

**Step 4**

Print the output of the show interface brief command.

### **Answer**

|  |
| --- |
| print ''  print 'Show Interface Output'  print '-----------------------------------------------------'  print ''  print output\_data  session.sendline('quit')  session.kill(0) |

## Create an Inventory Function

**Step 5**

Create a function that reads the device information from the file real-devices located in the PRNE/section11 folder. Return a list of devices, with information for each device stored in a dictionary.

### **Answer**

|  |
| --- |
| import pexpect  #-----------------------------------------------------------  def read\_devices\_info(filename):  devices\_list = []  file = open(filename,'r')  for line in file:  device\_info\_list = line.strip().split(',')  device\_info = {}  device\_info['name'] = device\_info\_list[0]  device\_info['ip'] = device\_info\_list[1]  device\_info['username'] = device\_info\_list[2]  device\_info['password'] = device\_info\_list[3]  devices\_list.append(device\_info)  return devices\_list |

**Step 6**

Create a function to connect to a device, taking as parameters the IP address, username, and password for each device, and returning a Pexpect session.

### **Answer**

|  |
| --- |
| # The following code connects to a device  def connect(dev\_ip,username,password):  print '--- connecting IOS: telnet '+dev\_ip  session = pexpect.spawn('telnet ' + dev\_ip, timeout=20)  result = session.expect(['Username:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0  print '--- attempting to: username: ' + username  # Successfully got username prompt, logging with username  session.sendline(username)  result = session.expect(['Password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0  print '--- attempting to: password: ' + password  # Successfully got password prompt, logging in with password  session.sendline(password)  session.expect('>')  return session # return pexpect session object to caller |

**Step 7**

Create a function which runs a **show interface summary** command on a device, taking as input a session object for a connected device.

### **Answer**

|  |
| --- |
| # The following function gets and returns interface information  def show\_int\_summary(session):  session.sendline('show interface summary')  result = session.expect('>')  show\_int\_summary\_output = session.before  return show\_int\_summary\_output |

**Step 8**

Create a function which prints information about a device, including device info (name, [IP](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP&type=terms), username, password), and the interface information for the device.

### **Answer**

|  |
| --- |
| # The following function prints device information  def print\_device\_info(device\_info,show\_int\_output):  print '-------------------------------------------------------'  print ' Device Name: ',device\_info['name']  print ' Device IP: ',device\_info['ip']  print ' Device username: ',device\_info['username'],  print ' Device password: ',device\_info['password']  print ''  print ' Show Interface Output'  print ''  print show\_int\_output  print '-------------------------------------------------------' |

**Step 9**

Create main application code which (a) reads device information from a file, (b) iterates through the list of devices, using your created functions to connect to each one, read interface information, and print nicely formatted output for each device.

### **Answer**

|  |
| --- |
| # Main program: connect to device, show interface, display  if \_\_name\_\_ == '\_\_main\_\_':  devices\_list = read\_devices\_info('real-devices')  for device\_info in devices\_list:  session = connect(device\_info['ip'],  device\_info['username'],  device\_info['password'])  if session == 0:  print '--- Session attempt unsuccessful ---'  continue  show\_int\_output = show\_int\_summary(session)  print\_device\_info(device\_info,show\_int\_output)  session.sendline('quit')  session.kill(0) |

# Chapter 12 Object Oriented Programming

# 12.1 Introduction

Object-oriented programming (OOP) has been used regularly by software developers for a couple decades. Network management systems have used object-oriented programming languages such as C++ and Java for just as long. This lesson provides an overview of the principles of object-oriented development, and the ways that it can be applied to networks and network programmability.

The figure above compares software that is developed using traditional, structured programming techniques, versus those created using object-oriented technology.

In a traditional, non-OO application, the following components are visible:

* **Global variables:** Typically structured applications will have a fair amount of data stored in 'global' storage, accessible by most, if not all, parts of the application. The danger with global variables is that they can be changed from anywhere, possibly causing unintended side-effects, and adding an element of complexity to applications.
* **Code:** Separating code into functions that are based on purpose is a good thing, and helps to isolate functionality and provide for code re-use. However, functions must have data passed to them, which can sometimes cause unnecessary complexity and overhead.
* **Main:** Every standalone application will have a 'main' set of code, that which gets executed from the start to the finish. Having code that is located in functions can help to keep the 'main' part of the application from getting too long and messy, but functions can only go so far in this endeavor.

In comparison, object-oriented software tends to be composed as follows:

* **Functions:** There will still be some independent, external functions, usually performing operations on smaller, isolated bits of information. Examples might be translation or MAC or IP addresses from strings to binary format, parsing of route strings, extracting version information, and so on
* **Code & data:** The unique aspect of object-oriented software development is that data and code will reside within 'objects'. The data (called attributes) will represent information and state regarding the object; the code will be object-based functions (called methods) which will perform operations on the object's data. Thus, all related data and code are organized in this manner, and held together in the object to which they belong.
* **Main:** Since there are less global variables (data is stored in objects), and fewer functions to call, object-oriented applications can have a simpler and smaller 'main' section of code.

These abstract ideas begin to paint the picture of what object-oriented programming is all about, but the question may be asked, "What is an object, and how would I use it?" The following example will help solidify the theory into a tangible, real-world environment.

**Object Oriented and Networks**

Object-oriented programming languages and techniques have been used by network management systems for more than two decades. Why is OO so popular among network control and monitoring systems? Basically because networks lend themselves directly to the use of OO technology and modeling.

Consider the network picture below:

* Networks: A collection of objects
  + Routers
  + Switches
  + Links, Interfaces, and Ports
* Each object:
  + Data (Attributes)
  + Operations (methods)
* Model-driven

Every item in the picture can be thought of as an 'object' from an object-oriented programming point of view. Specifically:

* Every switch has data that is associated with it, such as switch name, IP address, ports, spanning tree information, software version, and so on. Every switch also has certain operations that can be done to it: enable a port, disable a port, turn on edge security, set up ACLs, even download new software. Switch objects should be able to store much of this information, and provide methods for performing operations on that data.
* Every router has data that is associated with it such as router name, IP addresses, subnets and VLANs, interfaces, policy-based routing state and information, OSPF or IS-IS topology. And every router also has certain operations that can be done to it: enable an interface, disable an interface, set static routes, routing protocols, set up ACLs or policy-based routes, and so on.
* The internet itself can be thought of an object with attributes (access data, bandwidth, firewalling information), with operations (enable or disable firewall policies, set bandwidth levels for specific types of traffic, configuration of access parameters).

It is true that the actual devices themselves have the actual data, and have actual operations which can be performed. However, what was described above is the idea of model-driven control and management of systems. In this environment, software models of the actual devices are used to simplify the job of the network programmer. And through using these models (objects), the capabilities of a network programmability application can be increased significantly.

# 12.2 Terminology

One of the first things to understand about OOP is the terminology. The following terms are used for describing object-oriented functionality and data, and must be understood.

* **Classes versus Objects:** In OOP, the formal definition of a particular type of object is called a class. Thus you will define a class in your code, using the 'class' keyword. The class is the template of the actual object that you will later create. There will be one definition the class.

When you create an instance of that class, that instance is called an object. Creating an instance of an object is called instantiation. When you instantiate an object, you refer to the name of the class, which tells Python what type of object you wish to create. There will be zero, one, or many instances of an object.

For example, you may define a switch class. Then in your code, you will create some number of switch objects.

* **Attributes:** Attributes are the data associated with instances your class. Every object instance has its own data. Attributes are sometimes called variables, members, or member data. For example, you may define a switch class, and then instantiate some number of switch objects. Each of those objects will have its own data; each object with have its own switch name, software version, ports, and so on.

There are actually two types of attributes: class attributes and instance (or object) attributes. You will most always use instance variables, which are specific to the object instance in which they reside. However, there is also the lesser-known concept of a class variable, of which there is one per class. For most cases, instance attributes are used.

* **Methods:** Methods are the functions that you will call in order to perform some operation on your object. Operations can be as simple as getting or setting object attributes, or as involved as connecting to a device using some mechanism such as Pexpect or an SDN controller.

One of the most fundamental ideas of object-oriented programming is the idea that if you call a method on an object, you should not need to know about the details of how the object is going to accomplish the task that you give it. So for example, if your goal is to get the software version of the network device you are connecting to, then you should not have to worry about how the object gets that version information. It may connect via SSH and do a **show version brief**, or there may be another command required on a different device such as a wireless controller or AP. If you have created network device objects to take care of these details, then the rest of my application need not concern itself with how to deal with device type A or device type B.

# 12.3 Classes and Objects

The figure below gives a graphical view of the difference between classes and objects. On the left is the class definition itself, of which there is only one. On the right are the instantiated objects that were created, using the class definition as a template.

The figure shows class definition versus object instance. The class definition on the left will describe the attributes that are contained in the object, and the methods that have been provided to implement the functionality of the class.

The object instances on the right have been created by an application, and represent actual devices in the network. Each object instance has the same set of attributes (for example, 'name', 'ip\_address'), but each has its own attribute data (for example, the value of 'name' for the first device will 'ios-xr-1', the second would be 'ios-xr-2').

The object instances all have the same methods but these methods will operate on the data which is local to the specific object instance. Calling 'get\_name()' on the top object would return 'ios-xr-1', calling 'get\_name()' on the second object would return 'ios-xr-2', and so on.

# 12.4 Attributes (Variables)

The attributes for a class can be of any type—string, number, list, dictionary, complex combinations of data structures, and so on. They can even be other classes; for example, if you have a NetworkDevice class, you may also have an Interface class, and your NetworkDevice would have a list of Interface objects.

There are a few things to be aware of regarding object-oriented attributes in Python:

* **Created when used:** Languages with static typing require the programmer to explicitly define all attributes of a class. Because Python does not require declaration of variables, it likewise does not require the declaration of attributes. Instead, just like in the rest of the language, Python creates attributes only when they are first used. If the only time you use an attribute is in a specific method call, that attribute will not exist until that function is called. Therefore, it is a good practice to initialize all your attributes during the initialization of your object. That initialization is done in the '\_\_init\_\_' call, described next.
* **init():** When object instances are initialized, the \_\_init\_\_ function for your object is called. The \_\_init\_\_ function is where you have a chance to set up your object and is the place to initialize your instance attributes to their appropriate value.
* **'self':** Python requires that the first parameter for every method be 'self'. This 'self' parameter is a reference to this specific object instance.

When you reference an instance attribute, you must precede the attribute name with the name of your object instance, which is 'self'.

The example which follows helps to describe these facets of Python attributes for classes and objects.

#### Attributes Example

The example shows how a 'NetworkDevice' class might be constructed, showing both attributes and methods.

The code for this class definition is as follows:

class NetworkDevice:

node\_type = 'network'

def \_\_init\_\_(self,ip,os)

self.ip\_addr = ip

self.os\_type = os

def set\_credentials(self,user,pw)

self.username = user

self.password = pw

Things to note:

class NetworkDevice:

The keyword **class** identifies what follows as a class definition. The name that follows is the name of the class. By convention, class names use what is commonly called 'camel-case', meaning that the first letter of each word in the name is capitalized, without the typical underscores ('\_').

def \_\_init\_\_(self,ip,os):

def set\_credentials(self,user,pw):

The keyword **def** within a class definition is used to identify the methods of the class. In the first case, the method is the initialization method, which must be named '\_\_init\_\_'. Notice that for this method, and for the 'set\_credentials' method, the first parameter is always 'self', which is then used within the code to reference instance attributes.

self.ip\_addr = ip

self.os\_type = os

Code within each method will access instance attributes but prepending the attribute name with 'self.', for example, **self.attribute,** which tells Python that you are referencing an attribute which is unique to this specific object.

node\_type = 'network'

One last item – the example placed a class attribute into the class, to show the difference between class and instance attributes. You will use instance attributes far more often than class attributes, but it is important to remember that instance attributes must always have 'self.' preceding them.

# 12.5 Methods

Methods are used to perform operations, typically on the instance data for the object, sometimes to execute some external operation based on attributes of the object.

In the following code, observe the two aspects of methods that: (a) Each parameter list must have 'self' as the first parameter; and (b) Referencing instance attributes requires the use of 'self.' before the attribute name.

class NetworkDevice():

def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):

self.name = name

self.ip\_address = ip

self.username = user

self.password = pw

self.session = None

def get\_name(self):

return self.name

def get\_ip\_address(self):

return self.ip\_address

def connect(self):

self.session = pexpect.spawn('ssh '+self.username+

'@'+self.ip\_address,

timeout=20)

result = self.session.expect(['password:', pexpect.TIMEOUT])

...

self.session.sendline(self.password)

...

Observe the following from the code above:

* Initialization function (\_\_init\_\_) takes parameters include two with default values.
* 'self.' is used for referencing attributes of the object (instance attributes). Note that you will mostly be using either local variables (like 'result' in the example above) and instance variables (denoted with the 'self.' prefix
* The session object is local within the object – the calling code does not need to maintain this detail; it is the responsibility of the NetworkDevice object to maintain that information.

# 12.6 Creating Classes and Objects

## Classes for Storing Network Information

In this lab you will define a class, use that class definition to create objects to store information about networking devices, and print the information for those network device objects.

Note: For this exercise you will hard-code your device information as you create your network device objects.

**Step 1**

Define a class called NetworkDevice. Define a method within the class that takes device information as parameters: device name, [OS-](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS&type=terms)type, [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms), username, and password. Allow the username and password to be omitted, providing default values of 'cisco' and 'cisco'.

### **Answer**

|  |
| --- |
| class NetworkDevice():  def set\_info(self, name, os, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.os\_type = os  self.username = user  self.password = pw |

**Step 2**

Define a function to print a table of device information (name, OS-type,[IP](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP&type=terms), username, password) for every device. P

ass in a list of devices, where each device is an object of type NetworkDevice.

### **Answer**

|  |
| --- |
| #---- Function to go through devices printing them to table -----------  def print\_device\_info(devices\_list):  print ''  print 'Name OS-type IP address Username Password'  print '--------- ------- ---------- -------- --------'  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:11} {1:8} {2:12} {3:9} {4:9}'.format(device.name,  device.os\_type,  device.ip\_address,  device.username,  device.password)  print '' |

**Step 3**

Your 'main' code should create two or more NetworkDevice objects. For each object, call your method to set the device information.

Note: since you are hard-coding these devices, you are not reading from a file, or using a loop. Create the first object and set its info, then create the second object and set its info.

### **Answer**

After creating your device objects, add them to a Python list of devices. Call your print function, passing in the devices list.

|  |
| --- |
| #---- Main: read device info, then print ------------------------------  dev1 = NetworkDevice()  dev1.set\_info('dev1','IOS-NX','9.9.9.9')  def  dev2 = NetworkDevice()  dev2.set\_info('dev2','IOS-XE','8.8.8.8','chuck','secret')  print\_device\_info([dev1,dev2]) |

## Defining a Class with Information Set at Initialization

In this lab, you will define a network device class, with an initialization method for setting attributes for each created object.

You will read device information from two files – the PRNE/section12/devices and PRNE/section12/real-devices.

**Step 4**

Define a class called NetworkDevice. Define an initialization method (called \_\_init\_\_) within the class that takes device information as parameters (device name, OS-type, IP address, username, and password.

### **Answer**

Remember that 'self' must be the first parameter for every method.

|  |
| --- |
| #---- Class to hold information about a network device ----------------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, os, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.os\_type = os  self.username = user  self.password = pw |

**Step 5**

Create a function that takes the name of the devices file as input, reads the device information from the file, and creates network device objects, adding them to a list of devices. The result will be a list of network device objects, based on the information read from the file. The function should return the list of devices to the caller.

### **Answer**

|  |
| --- |
| #---- Function to read device information from file -------------------  def read\_device\_info(devices\_file):  devices = [] # Create a list for all devices  # Read in the devices from the file  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',') # Get device info into list  # Create a device object with this data  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[1],device\_info[3],device\_info[4])  devices.append(device) # add this device object to list  file.close() # Close the file since we are done with it  return devices # return a reference to the list we created |

**Step 6**

Create a print function that takes as input a list of network device objects, and prints a table of the devices from the list.

### **Answer**

|  |
| --- |
| #---- Function to go through devices printing them to table -----------  def print\_device\_info(devices\_list):  print ''  print 'Name OS-type IP address Username Password'  print '--------- ------- -------------- -------- --------'  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:11} {1:8} {2:16} {3:9} {4:9}'.format(device.name,  device.os\_type,  device.ip\_address,  device.username,  device.password)  print '' |

**Step 7**

Your main code will (a) call the function to read device information from the file, and (b) print the device information. It will do this twice; once for the 'devices' file, and once for the 'real-devices' file.

### **Answer**

|  |
| --- |
| #---- Main: read device info, then print ------------------------------  devices\_list = read\_device\_info('devices')  print\_device\_info(devices\_list)  devices\_list = read\_device\_info('real-devices')  print\_device\_info(devices\_list) |

Chapter 13 Object-Oriented Programming: Classes

# 13.1 Introduction

One of the more significant motivations for the move to object-oriented programming is inheritance. OOP attempts to model the 'real world', and in the real world there are types of things that inherit attributes from other types of things. For example, a circle is a type of shape, as is a rectangle or an octagon or oval. All these shapes have certain attributes in common, such as the ability to calculate the area of the shape..

In the networking world, there are network nodes. Different types of nodes would be hosts and network devices. Different types of network devices would be routers, switches, wireless access points, and so on.

This idea of certain types having 'child' classes, which inherit the attributes of its 'parent class, is fundamental to object-oriented programming. With Python, you can define classes, and sub-classes which inherit properties of the base or parent class.

This lesson will focus on how the OOP principle of inheritance is implemented using Python.

# 13.2 Inheritance

In inheritance, the basic idea is that there is a 'base' or 'parent' class, which represents the general data and functionality for the most general definition of that type of object. This parent class provides the basis from which child classes are derived. The derived classes inherit the attributes and functions of the base class, and they add their own unique attributes and functions which are particular to that child type of object.

The relationship between parent and child class is referred to as an "is-a" relationship, as in "a circle is a shape." In a networking context, "a router is a network device." Other networking examples include "a gigabit Ethernet interface is an interface," and "a link-state topology is a topology."

Consider the following snippets of code:

class NetworkDevice():

# general data: ip,user,pw

# general methods

class IOSDevice(NetworkDevice):

# IOS-specific data

# IOS-specific methods

class XRDevice(NetworkDevice):

# XR-specific data

# XR-specific methods

In the code above, notice:

* **Base class:** The class **NetworkDevice** is the base class for all types of networking devices. One can imagine child classes for various generic device types, such as router, switch, wireless device, firewall, [WAN](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=WAN&type=terms) optimizer, and so on.

The example is creating just one level of inheritance, and defining child classes which are either [IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS&type=terms)devices or IOS-XR devices.

The base class will contain attributes that are common to all children of its type. For example, all networking devices will have attributes for the device name, device management IP address, device administrator username, password, and so on. And there will likely be methods common to all networking devices, such as to provide the ability to set or retrieve these values.

* **Child classes:** The example defined two child classes, one for IOS devices, one for IOS-XR devices. As shown in the comments in the code, there will be attributes (data) and methods that are specific to each device type.

Child classes are derived from a base class, and the definition of the child class references its parent class in parentheses at the top of the class definition. In the example:

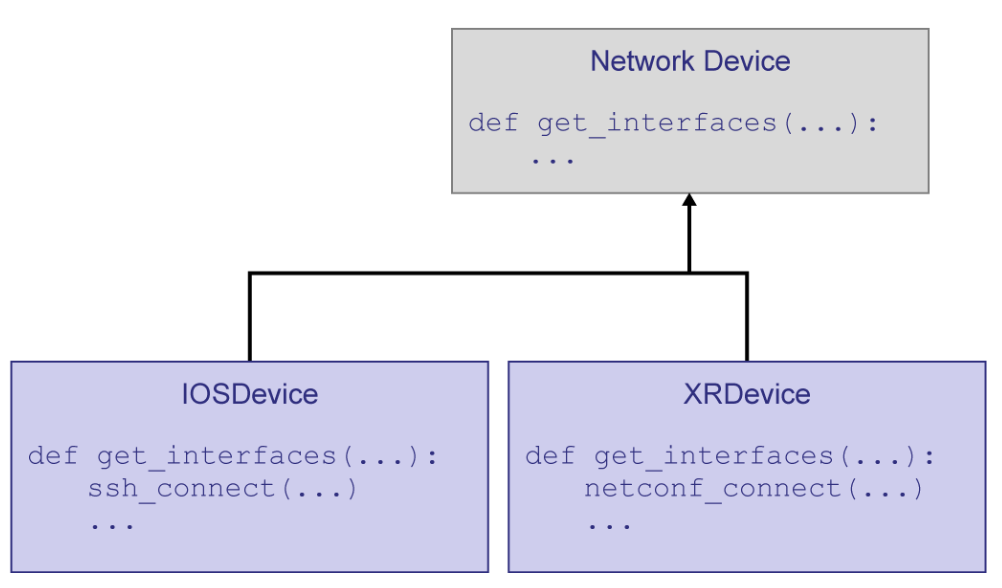
class IOSDevice(NetworkDevice):

The class IOSDevice above derives from the class NetworkDevice.

The class definition for each type of device may contain device-specific information (for example, IOS-XR routing capabilities, IOS switching capabilities). For the IOS-XR type of device, there may be routing attributes and the ability to call methods to set static routes. For the IOS type of device, there may be switching attributes and the ability to set parameters for spanning tree and edge security.

## Overriding Methods

The next important aspect of inheritance is the ability to override methods in each child class. The general idea is that a common method may be defined in the base class, but each child class will define device-specific implementations of that method.



Consider the following code:

class NetworkDevice():

def get\_interfaces(self,...):

return None

class IOSDevice(NetworkDevice):

def get\_interfaces(self,...):

session = ssh\_connect(...)

interfaces = show\_int\_brief(self.ip,...)

return interfaces

class XRDevice(NetworkDevice):

def get\_interfaces(self,...):

session = netconf\_connect(...)

interfaces = get\_netconf\_oper\_int(self.ip,...)

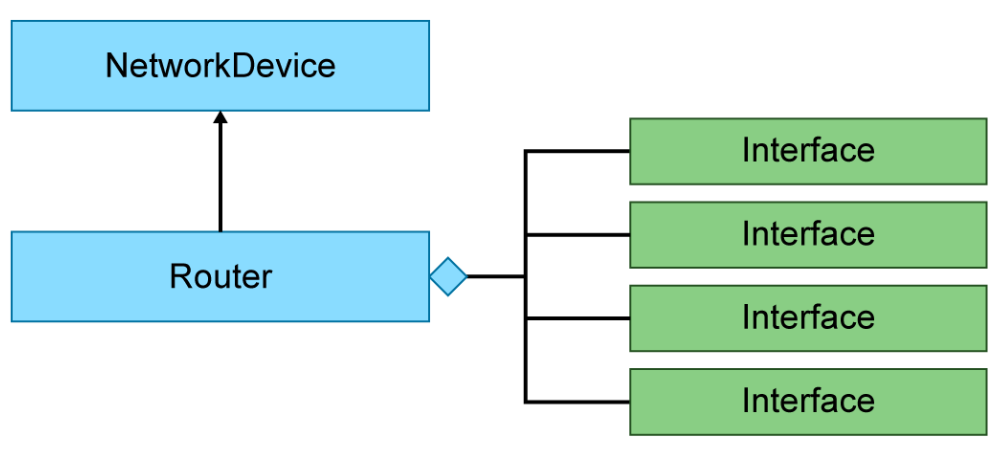
return interfaces

In the code above, the base NetworkDevice class defines a method called get\_interfaces(...). But each sub-class implements its own version of that method. In the simplistic example above, the IOS device implements get\_interfaces using [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH&type=terms), whereas the XR device, which supports [NETCONF](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=NETCONF&type=terms), is able to satisfy the request using that protocol.

# 13.3 Composition

Inheritance is a powerful feature of OOP, but overuse of this feature can be problematic. Therefore, before embarking on the user of inheritance in your classes, consider whether composition may be a preferable design for your classes.

As mentioned, inheritance represents an is-a relationship. An IOS-XR router is a network device. Composition, on the other hand, represents a has-a relationship. An IOS-XR router has a set of interfaces. An IOS-XR router has a set of static routes.



These examples are fairly straightforward. Some examples that may require careful thought might be the question if an XR router has an [OSPF](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OSPF&type=terms) state? Or is it an OSPF router?

Helping to understand the question may be whether your device would have to inherit from multiple parent classes, for example, if a router is both an OSPF router and an [IPv4](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IPv4&type=terms) router. Inheriting from multiple parent classes is allowed in Python, but quite often it can lead to confusing and complicated code, and thus should be considered with care.

# 13.4 lab Defining Classes

## Define Base and Child Classes

In this exercise, you will define a base class and child classes for different types of network devices.

**Step 1**

Define a base class for a generic network device, with an initialization function to set device name, IP, username, and password. Because the device type is unknown for a generic device, set os\_type to 'unknown'.

### **Answer**

|  |
| --- |
| #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  self.os\_type = 'unknown' |

**Step 2**

Define two child classes which derive from the base NetworkDevice class. One class will be for IOS devices, the other will be for IOS-XR devices. Each specific device class will have an initialization function which takes as input the device name, IP, username, and password. It will set its os\_type to the appropriate value.

### **Answer**

|  |
| --- |
| #---- Class to hold information about an IOS-XE network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios'  #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios-xr' |

**Step 3**

Create a function which reads the devices file located in the PRNE/section13 folder, creates the device objects, and adds them to the list. Your function will create different device objects depending on whether the os-type of the device that is read from the file is an IOS device, an IOS-XR device, or neither. If neither, your function will ignore the device and continue reading the file.

Your function should return a list of your created devices objects (note that some will be IOS objects, some will be IOS-XR objects.

### **Answer**

|  |
| --- |
| #---- Function to read device information from file -------------------  def read\_device\_info(devices\_file):  devices\_list = []  # Read in the devices from the file  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',') # Get device info into list  # Create a device object with this data  if device\_info[1] == 'ios':  device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  elif device\_info[1] == 'ios-xr':  device = NetworkDeviceXR(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  continue # go to the next device in the file  devices\_list.append(device) # add this device object to list  file.close() # Close the file since we are done with it  return devices\_list |

**Step 4**

Create a function which prints the devices list that was created.

### **Answer**

|  |
| --- |
| #---- Function to go through devices printing them to table -----------  def print\_device\_info(devices\_list):  print ''  print 'Name OS-type IP address Username Password'  print '------ ------- -------------- -------- --------'  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:8} {1:8} {2:16} {3:8} {4:8}'.format(device.name,  device.os\_type,  device.ip\_address,  device.username,  device.password)  print '' |

**Step 5**

Your main code should read the devices file, and print the device object information.

Your output should look something like the following:

|  |
| --- |
| Name OS-type IP address Username Password  ------ ------- -------------- -------- --------  d01-is ios 10.3.21.5 cisco cisco  d02-is ios 10.3.21.6 cisco cisco  d05-xr ios-xr 10.3.21.9 cisco cisco  d06-xr ios-xr 10.3.21.10 cisco cisco |

### **Answer**

|  |
| --- |
| #---- Main: read device info, then print ------------------------------  devices = read\_device\_info('devices')  print\_device\_info(devices) |

## Define Classes and Overriding Methods

In this exercise, you will define a base class for a generic networking device and sub-classes for IOS and IOS-XR devices. You will also override a method to have specific implementations of the method for each child class.

**Step 6**

Define a base class for a generic network device, with an initialization function to set device name, [IP](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP&type=terms), username, and password.

Note: for this exercise you will not be storing os\_type. You will be creating methods which return the[OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS&type=terms) type specifically for the base or child classes.

There will be a method for the base class called **get\_type** which returns **base**.

### **Answer**

|  |
| --- |
| #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  def get\_type(self):  return 'base' |

**Step 7**

Define two child classes which derive from the base **NetworkDevice** class. One class will be for IOS devices, the other will be for IOS-XR devices. Each specific device class will have an initialization function which takes as input the device name, IP, username, and password.

Create methods for each child class called **get\_type**. For the IOS class, this method will return 'IOS', for the XR class, this method will return 'IOS-XR'.

In your initialization method, rather than setting your attributes directly, you will call into the initialization method for your base class to set the attributes for name, IP, username, and password.

### **Answer**

|  |
| --- |
| #---- Class to hold information about an IOS-XE network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  def get\_type(self):  return 'IOS'  #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  def get\_type(self):  return 'IOS-XR' |

**Step 8**

Create a function which takes as input the name of the devices file located in the PRNE/section13 folder and creates device objects appropriate to the device os-type. Do not skip any devices; if it is an IOS-XR device, create an XR object; if it is an IOS device, create an IOS object; if it is neither, then create the generic, base type of device.

Your function will return a list of your created devices objects; some will be IOS objects, some will be XR objects, and some will be base objects.

### **Answer**

|  |
| --- |
| #---- Function to read device information from file -------------------  def read\_device\_info(devices\_file):  devices\_list = []  # Read in the devices from the file  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',') # Get device info into list  # Create a device object with this data  if device\_info[1] == 'ios':  device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  elif device\_info[1] == 'ios-xr':  device = NetworkDeviceXR(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device) # add this device object to list  file.close() # Close the file since we are done with it  return devices\_list |

**Step 9**

Create a function which prints the devices list that was created.

### **Answer**

|  |
| --- |
| #---- Function to go through devices printing them to table -----------  def print\_device\_info(devices\_list):  print ''  print 'Name OS-type IP address Username Password'  print '------ ------- -------------- -------- --------'  # Go through the list of devices, printing out values in nice format  for device in devices\_list:  print '{0:8} {1:8} {2:16} {3:9} {4:9}'.format(device.name,  device.get\_type(),  device.ip\_address,  device.username,  device.password)  print '' |

**Step 10**

Your main code should read the devices file, and print the device object information.

Your output should look similar to:

|  |
| --- |
| Name OS-type IP address Username Password  ------ ------- -------------- -------- --------  d01-is IOS 10.3.21.5 cisco cisco  d02-is IOS 10.3.21.6 cisco cisco  d03-nx base 10.3.21.7 cisco cisco  d04-nx base 10.3.21.8 cisco cisco  d05-xr IOS-XR 10.3.21.9 cisco cisco  d06-xr IOS-XR 10.3.21.10 cisco cisco  d07-xe base 10.3.21.19 cisco cisco  d08-xe base 10.3.21.22 cisco cisco |

### **Answer**

|  |
| --- |
| #---- Main: read device info, then print ------------------------------  devices = read\_device\_info('devices')  print\_device\_info(devices) |

# 13.5 Encapsulation

One of the concepts of OOP is the principle of encapsulation, also known as data hiding. The general idea of the principle is that code which uses the objects you create should not need to know about the internal details and workings of your object – the code should be able to access your methods to perform whatever functionality that method offers.

Some languages take the idea of encapsulation a step further, and make it possible to absolutely prohibit other code from accessing or changing internal items within an object. These private attributes and private methods cannot be accessed by any code other than the code of the object itself.

In Python, there are no 'private' attributes or members. Every attribute of the object you have created is visible and accessible to the outside world. If this is a problem, Python provides workarounds to help achieve the hiding of your data:

* **Properties:** Python provides the concept of making your data into a 'property', which makes it inaccessible to outside code. There are two methods of doing this:
  + **property(..) function**: Using the property function, it is possible to declare certain attributes as only accessible via getter and setter methods, which allow outside code to access the attribute via methods that are implemented in the object – insuring that the attributes are protected from invalid values.
  + **@property decorator**: Using the property decorator, it is possible to declare attributes as only accessible via a getter function by the same name. The decorator also allows for setting of the attribute.
* **Mangling:** It is also possible to 'mangle' the name of your attributes, by prefacing the attribute name with double underscores. Mangling tells Python that the attribute is intended to be private, and not directly accessible by outside code.

Neither of these methods entirely secures your classes from malicious outside access, but they do help prevent some inadvertent problems that might arise.

# 13.6 lab Defining Child Classes

In this lab, you will define a base class for a generic network device, and will define child classes for specific device types:[IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS&type=terms) and IOS-XR. The main functionality for this lab will be to implement **connect** and **get\_interfaces** methods for your child classes.

You will read in the **challenge-devices** file that is located in the **~/Desktop/PRNE/section13** folder. This file has the following information:

ios-01,ios,10.30.30.1,admin,cisco

ios-02,ios,10.30.30.2,admin,cisco

xr-01,ios-xr,10.0.0.1,cisco,cisco

base-01,unknown,10.20.20.1,unknown,unknown

Lab is currently initializing: 26% complete

## Defining Child Classes

**Step 1**

Define a base network device class with the usual attributes for name, [IP](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP&type=terms), username, and password. Your base class will have an empty connect method that sets the session attribute to **None**, and a **get\_interfaces** method that returns a string such as "Base device, cannot get interfaces".

### **Answer**

|  |
| --- |
| import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  self.interfaces = ''  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---' |

**Step 2**

Define child device-specific classes for [IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS&type=terms) and IOS-XR device types.

For your IOS class, your connect method will actually connect to the device, storing the session as an attribute. Your **get\_interfaces** method will do run the **show int brief** command on the device, and will store the output.

For your IOS-XR class, you will create 'stub' methods for both **connect** and **get\_interfaces**. The **connect**method will set the session to None, and the **get\_interfaces** method will return a string saying "Getting interface information a different way", to simulate the behavior of different object types satisfying requests in a device-specific manner.

### **Answer**

|  |
| --- |
| #---- Class to hold information about an IOS-XE network device --------  class NetworkDeviceIOS(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return None  result = self.session.expect(['Password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return None  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before    #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting XR: ssh '+self.username+'@'+self.ip\_address  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.interfaces = '--- XR Device interface info ---' |

**Step 3**

Create a function to read in the 'challenge-devices' file, creating and returning the list of device objects, created as appropriate for each IOS, IOS-XR, and generic device in the file.

### **Answer**

|  |
| --- |
| #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',')  # Create a device object with this data  if device\_info[1] == 'ios':  device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  elif device\_info[1] == 'ios-xr':  device = NetworkDeviceXR(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device)  return devices\_list |

**Step 4**

Create a function to print the device information (name, IP, username, password), and then display the interface information.

**No**

Your 'interfaces' for this exercise is only the complete string returned by the device via the **show interface summary** command. You do not need to parse that output, since this challenge is about OOP and not concerned with details of parsing CLI output.

### **Answer**

|  |
| --- |
| #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print ''  print ' Interfaces'  print ''  print device.interfaces  print '-------------------------------------------------------\n\n' |

**Step 5**

Your main code should read the devices file using your 'read' function, and for each device object, perform a connect, then perform a get\_interfaces, then print the device data using your 'print' function.

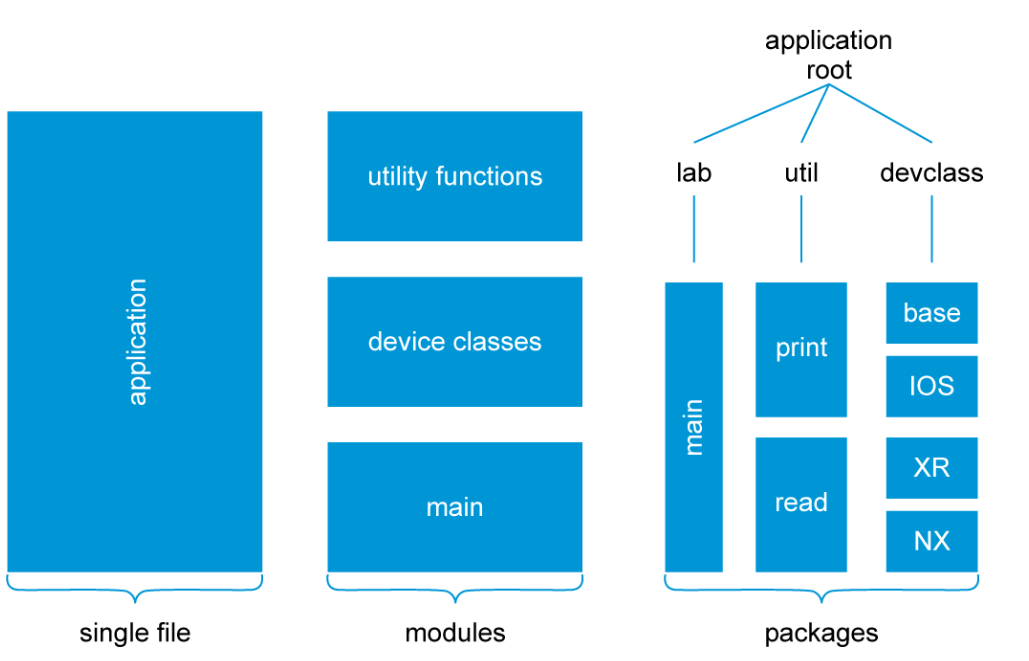
### **Answer**

|  |
| --- |
| #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('challenge-devices')  for device in devices\_list:  print '==== Device ============================================================='  session = device.connect() # Connect to device  device.get\_interfaces() # Get interface info for this device  print\_device\_info(device) # Print interface info for this device |

Chapter 14 Modules and Packages

# 14.1 Introduction

Your network programmability code will eventually grow to the point where having single files holding your scripts and applications no longer makes sense. At this point, you will want to organize your code into Python modules and packages.



The image shows the three different ways in which Python applications can be structured.

## Single Application File

On the left is a single application file. This format has the advantage of having all code located in the same file for easy access. However once your application becomes larger, the single file becomes crowded and can be confusing, hard to modify and even harder to maintain. Therefore, once your application begins to grow, you will want to separate the different pieces of functionality into modules.

## Modules

A Python module is technically just a file with Python code in it. Your single application is technically a 'module'. However, when people speak of separating an application into modules, the idea is to take common pieces of functionality and organizing them together into their own, separate Python files. These files are then called Python modules, which are a collection of Python functions or classes, which are grouped into a single file.

Your application will reference these modules, and the functions or classes within them, using the Python import statement.

## Packages

A Python package is a directory that contains Python modules. When your application becomes large enough to require more organization than can be provided by modules alone, you will want to break it apart into packages.

A directory becomes a Python package by having a special \_\_init\_\_.py file located within it. Every directory that is intended to be a package must have an \_\_init\_\_.py file in it. Your application will reference the modules within your packages in the normal way, using the Python import statement. Since you are running code from different directory, you will need to make sure that Python knows where to look for your packages, using the PYTHONPATH environment variable.

Modules and packages are important not only for organizing your own application, but for creating code that can be imported and called by other developers as well. The Python libraries that you have been using are all Python modules, hosting functions or classes that you will be using regularly in your network programming projects. So the value of modules and packages may go well beyond your own specific application.

Note: The remainder of these lessons will be referring to functions inside Python modules. It is also possible to define classes within modules – in fact, this is done in the examples. For purpose of clarity, this section will refer to functions, since that is the most common and simplest type of Python object to define in a module. If you read 'function', you should assume it to include classes as well.

# 14.2 Modules

Python modules are just regular Python files that happen to have several related functions or classes located within them. You use modules when your code becomes large enough that a single application file is no longer practical or desirable for holding your application code. In these situations, you will want to separate your code into modules.

There are two general categories of files to discuss when describing Python modules.

* **Module files:** You will group functions or classes according to your own organizational needs and preferences. The module files that you create will be regular Python code, with several functions or classes that are defined within. Other parts of your application (the callers) will call your functions or reference your classes, just as they would if they had been defined within the same application file.
* **Calling files:** Your application code will call the functions or classes in your modules just as they would have if they had been defined inside the main application file itself. The one difference is that the calling code will need to import your module and functions or classes using the Python import statement. This statement tells Python where to look for function and class definitions that are not located within the file of the calling code.

The following snippets of code show the definition of the module file, and the referencing of functions or classes in that file by the calling code. The module file ('util.py' in this case) has a utility function to read device information from a file, returning a list of devices. It also has a function to print device info.

File util.py:

#======================================================================

def read\_devices\_info(devices\_file):

...

return devices\_list

#====================================================================

def print\_device\_info(device):

print ...

print device.interfaces

The main application file will call these functions, just as if they were located within the same Python application file, with the exception that you need an import statement to let Python know where to find these functions and classes that are being referenced:

File main.py:

import util

...

devices\_list = read\_devices\_info('devices')

...

for device in devices\_list:

....

print\_device\_info(device)

# 14.3 Creating a Module

Python modules are simply files with Python code in them. The difference between a module and an application file is that the module file is intended to be called by the application. As such, the module file will contain functions and/or classes, typically related to each other in some way. For example, a module for utility routines, a module for a specific type of Python classes, a module for commonly used classes, and so on.

**Module Data**

When creating a Python module, especially with functions, it is important to consider carefully the variables you will be using in your functions. These variables will fall into three categories:

* **Parameters:** Parameters are the values passed to your function by the calling routine. Your formal parameters, which are enclosed in the parentheses in your function definition, are accessible to you and valid within that function only.
* **Local variables:** Local variables are those variables and data structures you define within your function. They are not seen by the calling routine, or by any other functions; they are local to your function alone.

**Note**

If you create a local variable, such as, a device list, and then you return that value to the caller using the return statement, then your local variable will be, in effect, passed back to the caller, and the calling function now has a reference to that data.

* **Global variables:** It is possible – though not necessarily a good idea – to define global variables for your module. These global variables are accessible and seen only by code within the module – calling code cannot see these variables. The scope of these variables is for the entire module.

If you create a local variable, such as a device list, and then you return that value to the caller using the return statement, then your local variable will be, in effect, passed back to the caller, and the calling function now has a reference to that data.

Classes have similar categories of data as in a module of functions, but since classes tend to have self-contained instance data, it can be less confusing. You can still have global data, however, which is accessible to all classes in the module.

**Module Code**

The main purpose of a module is to hold functions or classes that can be called by other pieces of code, either a main application or by code in another module which itself has been called by the main application. However, it is possible to define code in your modules which does not belong to any function or class – in other words, the code is not within any of your function or class definitions. This code can be considered global to the module; it is executed once, and only once, when the module is first imported by the application or by any other module.

If the purpose of your module is to hold some data, in addition to providing functions to be called, then having this type of module code might be useful to initialize that global data for the module. Using a module in this manner with a module containing only functions makes that module similar to a Python class, providing both data (like attributes) and functions (like methods). The major difference lies in the fact that for classes you will create separate instances, with their own instance data; this is not true for modules. The bulk of your code in a module will be located in your functions, but there may be some cases where non-function code may serve a useful purpose.

**Modules with Classes**

For modules with classes, the data is structured into class attributes, and the same rules apply within a class that is defined in a module, as with classes defined in the main application file itself. For modules of classes, you can have global variables as well, but object-oriented programming is intended to limit or remove the use of global variables, and should be avoided in modules as well.

# 14.4 Importing a Module

Your application must inform Python about the module you intend to use, in order to actually call a function, or reference a class within that module.

Consider the following sample code:

import devclass

from util import read\_dev\_info

...

dev\_list = read\_dev\_info('devices')

...

for device in devices\_list:

dev = devclass.NetworkDevice(...)

...

The example above shows multiple ways to import a module:

* **import module**: In the simplest format, you only need to include an **import** statement, informing Python that you intend to reference functions or classes from that module within your code. Using the **import** statement makes all functions or classes within the module available to be called.

When you import just using the name of the module, you must precede the function or class name with the name of the module in which it resides. In the example above, the code specifies only **import devclass**, which means that functions or classes that are referenced from that module must be preceded by the name of the module.

Importing the entire module may be a convenient method of allowing access to functions or classes within a module. However, consider that Python must make all functions or classes available, which may be inefficient. If the module has many functions or classes, it may be more efficient to use the next method.

* **from module import name**:In this more precise format, you specify both the module, and the actual function within the module you wish to reference. If your code needs to reference more than one function within a module, you will need to specify each of them, in order for Python to know your intended usage.
* **import module as name**: If for some reason your code already contains a variable or function with the same name as the function you intend to call from another module, you can change the local name by which you will refer to your module name.

# 14.5 Using Modules

In this lab, you will create modules and import them into your application.

Lab is currently initializing: 26% complete

## Using a module

In this exercise, you, will move functionality out of your main application file, and into Python modules. You will use the **import** statement to import the entire modules.

**Step 1**

Start with the file **old-main.py** which is located in the **~/Desktop/PRNE/section14/14-1** folder and which contains all application code for getting information from devices, including utility routines and classes.

**Step 2**

Separate out the utility functions **read\_devices\_info** and **print\_device\_info** and put them into a Python module that is named **util.py**.

### **Answer**

|  |
| --- |
| import devclass  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',')  # Create a device object with this data  if device\_info[1] == 'ios':  device = devclass.NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])    else:  device = devclass.NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device)  return devices\_list  #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print ''  print ' Interfaces'  print ''  print device.interfaces  print '-------------------------------------------------------\n\n' |

**Step 3**

Separate out the class definitions for your base network device and [IOS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IOS&type=terms) device, and put them into a Python module that is named **devclass.py**.

### **Answer**

|  |
| --- |
| import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address    self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])    # Successfully got username prompt  print '--- username:',self.username  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  result = self.session.expect('>')    # check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0    def get\_interfaces(self):  self.session.sendline('show interface summary')  result = self.session.expect('>')  self.interfaces = self.session.before |

**Step 4**

Modify your main application to import your modules using the import statement, and call them in order to get the interface information for each device.

**Note**

Depending on how you structure your utility function, you will likely need to import your device class module from your utility module.

### **Answer**

|  |
| --- |
| import util  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = util.read\_devices\_info('devices')  for device in devices\_list:  print '==== Device ============================================================='  device.connect()  device.get\_interfaces()  util.print\_device\_info(device) |

## Using a module and importing function into the main program

In this exercise, you will move functionality out of your main application file, and into Python modules, and will use the **from module import name** statement to import the entire modules while specifying which functions you would like to be defined in the main program name space.

**Step 5**

Start with the file **old-main.py** which is located in the **~/Desktop/PRNE/section14/14-2** folder, and which contains all application code for getting information from devices, including utility routines and classes.

**Step 6**

Separate out the utility functions **read\_devices\_info** and **print\_device\_info** and put them into a Python module that is named **util.py**.

### **Answer**

|  |
| --- |
| from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',')  # Create a device object with this data  if device\_info[1] == 'ios':    device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device)  return devices\_list  #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print ''  print ' Interfaces'  print ''  print device.interfaces  print '-------------------------------------------------------\n\n' |

**Step 7**

Separate out the class definitions for your base network device and IOS device, and put them into a Python module that is name **devclass.py**

### **Answer**

|  |
| --- |
| import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address    self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])    # Successfully got username prompt  print '--- username:',self.username  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  result = self.session.expect('>')    # check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0      def get\_interfaces(self):  self.interfaces = '--- IOS Device, getting interfaces a different way ---'  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before |

**Step 8**

Modify your main application to import specific functions and classes from your modules using the from module import name statement, and call them in order to get the interface information for each device.

### **Answer**

|  |
| --- |
| from util import read\_devices\_info  from util import print\_device\_info  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('devices')  for device in devices\_list:  print '==== Device ============================================================='  device.connect()  device.get\_interfaces()  print\_device\_info(device) |

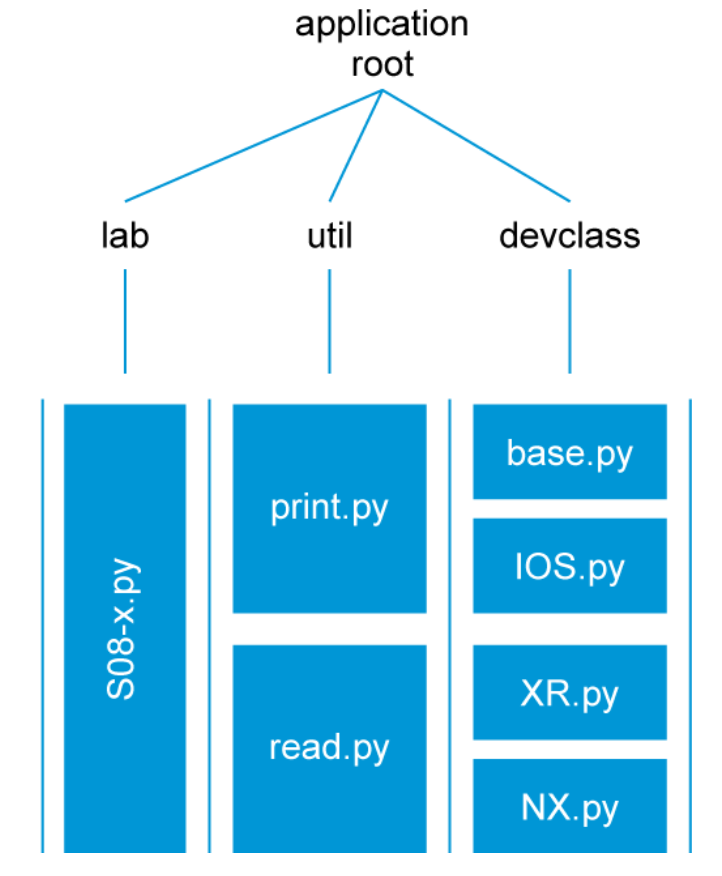
# 14.6 Packages

You will use packages when your project is large enough to warrant further organization of your application code. Packages are basically modules divided into separate folders or directories.

**Note**

Some operating systems refer their file system as containing 'folders', others use the term 'directories'. We will use 'directory' in this lesson.

* **Directories:** Packages use directory structure to organize code and modules
* Directories must have a file called **\_\_init\_\_.py** to identify as a package
* **Package Paths:** use PYTHONPATH environment variable and dots to specify path to modules



In the figure above, the application actually consists of Python code modules located in three different directories – 'lab', 'util', and 'devclass'. The 'lab' folder holds the main application code; the 'util' directory holds the utility modules for printing and reading; and the 'devclass' directory holds the modules for defining different device classes for use by the application.

The major points to know about using packages in Python:

* **Directories:** You use directories to organize your code modules.
* **\_\_init\_\_.py:**Every directory that is to become a Python package must have a file called \_\_init\_\_.py. This file can be completely empty, which is often the case. But it must exist.
* **Paths:** Python must know where to look on your system to find the modules you reference. If they are located in a Python package, then you must tell Python where to look. Your options are:
  + **Current directory:** If you are running your main application from a directory, and your Python package directories are located in that same directory, Python will be able to find them automatically.
  + **PYTHONPATH:** If your packages are located in another directory not directly in your application's runtime directory, you will need to set, and export, the PYTHONPATH environment variable. Python will use this variable to find your packages.

14.7 Creating a package

Creating a package involves two items:

* **Modules:** You will place your module files into the appropriate directory for your package.
* **\_\_init\_\_.py:** You will create an \_\_init\_\_.py file in your directory for your package.

Once these steps are complete, you are ready to begin using your Python package.

Consider the example Python application and package directory structure below:

S08-3/

main.py

---util

--\_\_init\_\_.py

--readdevs.py

--printdevs.py

---devclass

--\_\_init\_\_.py

--basedev.py

--iosdev.py

--xrdev.py

In the example there are two directories:

* **util**: The directory that holds the Python package for the utility function modules. Note the presence of \_\_init\_\_.py.
* **devclass**: The directory that holds the Python package for the device class definitions. Note the presence of \_\_init\_\_.py.

For this example, the main application (main.py) is located in the root directory. This means that Python is able to look in the child directories 'util' and 'devclass' in order to satisfy its search for Python packages when main.py reference modules in those packages. The PYTHONPATH environment variable is not required.

Consider another example:

S08-3/

---labs

--main.py

---util

--\_\_init\_\_.py

--readdevs.py

--printdevs.py

---devclass

--\_\_init\_\_.py

--basedev.py

--iosdev.py

--xrdev.py

In the example, the main.py application has been moved into its own directory. There are now three directories:

* **labs:** Where the main application lives. Notice that there is no \_\_init\_\_.py file – it is not required. Only package folders require this file.
* **util:** The directory that holds the Python package for the utility function modules. Note the presence of \_\_init\_\_.py.
* **devclass:** The directory that holds the Python package for the device class definitions. Note the presence of \_\_init\_\_.py.

Because the application code was moved into a different directory, the package directories for util and devclass are no longer located in the directory of the main application. Therefore, the PYTHONPATH needs to be specified to let Python know where to look for the modules.

In order to successfully run the application in this environment, you would need to do the following:

.../S08-4/labs$ export PYTHONPATH=~/PRNE/labs/mod08/S08-4/

.../S08-4/labs$ python main.py

14.8 **Using Packages**

In these exercises, you will move your modules into directories, making them packages, and referencing them from your main application's directory.

## Creating Packages from Modules

**Step 1**

Start with the modules **util.py** and **devclass.py** located in the **~/Desktop/PRNE/section14/14-3** folder. Move those modules into directories you create which you will use as Python packages. The directories should be created from within your main application directory.

### **Answer**

Remember to create your **\_\_init\_\_.py** file in your directories, in order to make them Python packages.

**Step 2**

Take the **util.py** file and create two Python files, one for reading and one for printing.

### **Answer**

Package: **util**; File: **util/readdevs.py**

|  |
| --- |
| from devclass.basedev import NetworkDevice  from devclass.xrdev import NetworkDeviceXR  from devclass.iosdev import NetworkDeviceIOS  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  file = open(devices\_file,'r')  for line in file:  device\_info = line.strip().split(',')  # Create a device object with this data  if device\_info[1] == 'ios':    device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  elif device\_info[1] == 'ios-xr':  device = NetworkDeviceXR(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device)  return devices\_list |

Package: **util**; File: **util/printdev.py**

|  |
| --- |
| #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print ''  print ' Interfaces'  print ''  print device.interfaces  print '-------------------------------------------------------\n\n' |

**Step 3**

With your device class package and module, make separate Python files for each of your device classes.

### **Answer**

Package: **devclass**; File: **devclass/basedev.py**

|  |
| --- |
| #======================================================================  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---' |

Package: **devclass**; File: **devclass/xrdev.py**

|  |
| --- |
| #======================================================================  import pexpect  from devclass.basedev import NetworkDevice  #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting: ssh '+self.username+'@'+self.ip\_address  self.session = pexpect.spawn('ssh '+self.username+  '@'+self.ip\_address, timeout=20)  result = self.session.expect(['password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  self.session.expect('#')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before |

Package: **devclass**; File: **devclass/iosdev.py**

|  |
| --- |
| #======================================================================  import pexpect  from devclass.basedev import NetworkDevice  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  #---- Initialize --------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  #---- Connect to device ------------------------------------  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address    self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])    # Successfully got username prompt  print '--- username:',self.username  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  result = self.session.expect('>')    # check for failure  if result != 0:  print '--- Timeout or unexpected reply from device'  return 0    #---- Get interfaces from device -----------------------------  def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before |

**Step 4**

Run your **main.py** application from the root directory – hence no need to set **PYTHONPATH**.

### **Answer**

|  |
| --- |
| from util.readdevs import read\_devices\_info  from util.printdev import print\_device\_info  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('devices')  for device in devices\_list:  print '==== Device ============================================================='  device.connect()  device.get\_interfaces()  print\_device\_info(device) |

## Using Custom Packages

**Step 5**

Take your main application file, and move it into its own directory called **lab**. Make sure you create this directory.

### **Answer**

When you execute your code from this directory, your packages will no longer be located off the directory where it is running, and hence you will need to set and export the **PYTHONPATH** variable.

|  |
| --- |
| cisco@ubuntu:~/Desktop/PRNE/section14/14-3/lab$ **export PYTHONPATH=~/Desktop/PRNE/section14/14-3:$PYTHONPATH**  cisco@ubuntu:~/Desktop/PRNE/section14/14-3/lab$ **python main.py** |

14.9 Importing a Module from a Package

Importing a module from a package is no different than importing a module, with the exception that you may need to reference the path to the package in your application code.

Chapter 15 Python and Data Storage

15.1 Introduction

Introduction

Once your network programmability application matures beyond the initial testing and prototype state, you will likely want to utilize some type of storage mechanism beyond an unstructured text file. This section discusses various mechanisms provided by Python for storing your networking data.

In general there are four categories of data storage in Python:

* **Unstructured Text:**A basic text file following no specific format. Easy to iterate in a **for** loop. These types of files can be used for prototyping and testing your code. When you are building 'production' code, you will probably want to use a structured text file of some type.
* **Structured Text:** There are tools that are provided in Python for dealing with structured files as well. Popular formats include comma-separated-value (CSV), JSON, and XML. Libraries exist for reading and writing these files, and converting them to and from typical Python data structures.
  + **CSV:** Compact table-like data with items separated by commas, each line in the file being like a row in a table or spreadsheet. Use these files for storing lists of homogenous data, such as a list of devices, log files, or something such as time series data.
  + **JSON:** Simple human-readable format for storing arrays (like Python lists) and objects (like Python dictionaries). These types of files are useful for storing dictionary-type data, with names (keys) and values. Used for storing data structures which may vary, requiring names for each of the data items. Device information which may vary from device type to device type would be a good candidate for a JSON file. JSON however is very popular and efficient, and so can be used for other types of data such as logs, time-series data, and just about any other type of data.
  + **XML:** Highly structured, less human-readable than JSON. XML can benefit from a data definition file which helps translate its data. XML was once popular but in many cases, XML has been replaced by JSON.
* **Binary:** Your application may need to deal with binary data as well, for things such as software images or certificates. Python provides mechanisms for dealing with these types of files as well. Binary files can be useful for storing binary data (software images, certificates), and also for storing general data more efficiently, such as time-series data representing numeric values. In such cases, rather than storing the information in a structured textual file, storing it in binary may be a faster and more efficient option.
* **Database:** SQL databases for storing tables, large volumes of data, and for performing complex searches. At some point, you may desire the use of a real database for storing large amounts of data, for searching, and for normal relational database operations such as maintaining tables of data and relationships between items in those tables. Python libraries exist for performing create, read, update, and delete SQL database operations on such data.

## Python with Statement

When dealing with files, there is a fair amount of overhead that is involved with opening and closing a file. The **with** statement simplifies file operations.

Normally you would perform an **open** to open the file, and a **close** to close it when finished:

file = open('myfile','r')

... # do something with the file

file.close()

Using the **with**statement:

with open('myfile','r') as file:

The **with** statement will take care of closing the file for you when you exit the code block. In the examples of these lessons, you will see the Python **with** statement in action.

# 15.2 CSV Files

Comma separated value files are read and written using the [CSV](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CSV&type=terms) library. The processes for reading and writing are as follows:

**Writing lists**(using Python list of lists)**:**

* Open the file**:**Open the file for writing as for a regular text file.
* Attach CSV writer: Create a CSV writer and attach it to the file.
* Write rows: Use the CSV writer object to write rows of data from your lists.

**Writing dictionaries**(using Python list of dictionaries)**:**

* Open the file: Open the file for writing as for a regular text file.
* Attach CSV dictionary writer: Create a CSC dictionary writer object and attach it to the file.
* Write header: Use the CSV dictionary writer to write the header – the keys of the dictionary.
* Write rows: Use the CSV dictionary writer to write the rows of dictionary data from your lists.

**Reading lists**(into a Python list of lists)**:**

* Open the file**:**Open the file for reading as for a regular text file.
* Attach CSV reader: Create a CSV reader and attach it to the file.
* You now have a Python list of lists (the actual CSV reader object) which can be used for iteration and for other normal list operations, and the items within the list are themselves normal Python lists which can be used with normal Python list operations.

**Reading dictionaries**(into a Python list of dictionaries)**:**

* Open the file: Open the file for reading as for a regular text file.
* Attach CSV dictionary reader: Create a CSC dictionary reader and attach it to the file.
* You now have a Python list of dictionaries (the actual CSV dictionary reader object), which can be used for iteration and other normal list operations, and the items within the list are dictionaries, which can be used with normal Python dictionary operations.

Here is an example of reading a CSV file. Reading a normal text file is straightforward in Python, since opening a text file for reading provides an object that is iterable:

file = open('myfile','r')

for line in file:

# 'line' must be parsed for specific data items

The problem with the above is that 'line' is just an unstructured string, which you will have to parse in some way in order to get data that makes sense.

Reading in the lines from a CSV file is similarly straightforward, with the advantage that the data from each row (line) is automatically parsed and placed into a list.

First you import the csv library for performing functions on CSV files:

import csv

You will open the file just as you would any other text file:

file = open('csv\_file', 'r')

You then use the csv library to create a CSV object for reading the file, which provides a csv variable which you can use for reading the file:

csv\_input = csv.reader(file)

Iterating through the CSV file looks similar to what is done for a text file, but with the added benefit that the line itself has already been parsed into a list:

for items in csv\_input:

# data is already parsed into list

The complete example to read in a list of device information:

file = open('csv-devices','r')

csv\_devices = csv.reader(file)

for device\_info in csv\_devices:

# device\_info is already parsed into a list of items

The main advantage is the ability to immediately have parsed items from each 'row' placed into your Python list, as you iterate through all rows in the file.

Here is an example of writing a Python file. Once again, the file is opened as you would with any normal text file:

outfile = open('csv-devices-out', 'w')

csv\_devices\_out = csv.writer(outfile)

For the output example, you will be writing a list of lists. For example, a list of devices, where every individual device is itself a list of device information (device name, [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms), [OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS&type=terms)-type, username, and password).

The normal operation would then be just to perform the **writerows** function, passing in the list of devices:

csv\_devices\_out.writerows(devices\_out\_list)

In some of the code and lab examples, you will see the use of the **with** statement. The preceding example using **with** would be as follows:

with open('csv-devices-out','w') as outfile:

csv\_devices\_out = csv.writer(outfile)

csv\_devices\_out.writerows(devices\_out\_list)

This example required the same number of lines, but recall that **outfile** will automatically be closed when the code block is completed. And the indentation of the code block improves readability, identifying the lines where the file is being accessed.

# 15.3 JSON Files

The main[JSON](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=JSON&type=terms) data structures, Object and Array, and very similar to the Python data structures of Dictionaries and Lists:

JSON Object and Python Dictionary:

* In JSON, an Object is a collection of name-value pairs, which are enclosed by curly brackets and separated by a colon: **{"name":"xr-01", "ip":"10.0.0.1"}**
* In Python, a Dictionary is a collection of name-value pairs, which are enclosed by curly brackets and separated by a colon: **{'name':'xr-01', 'ip':'10.30.30.1'}**

JSON Array and Python List:

* In JSON, an Array is a sequence of items of any type, which is enclosed by square brackets.
* In Python, a List is a sequence of items of any type, which is enclosed by square brackets.

You will convert from a JSON string into the appropriate Python structures using the **json.loads()** function. You will convert from a Python structure to a JSON string using the **json.dumps()** function.

Conversion is actually to and from a string. You will use a normal file read operation to read the JSON data from a file into a string, which you will then convert to Python data. And you will use a normal file write operation to write the JSON string to a text file.

The read operation works as follows:

json\_file = open('json\_devices', 'r') # just a normal file open

json\_device\_data = json\_file.read() # just a normal file read operation

devices\_list = json.loads(json\_device\_data) # convert to Python data structure

At this point, devices\_list is in the familiar structure which will follow what was defined in the JSON file itself. If each piece of information in the file was specified as a JSON object, as follows:

[

{"name":"ios-01","os":"ios","ip":"10.30.30.1","user":"cisco","password":"cisco"},

{"name":"ios-02","os":"ios","ip":"10.30.30.2","user":"cisco","password":"cisco"},

{"name":"ios-03","os":"ios","ip":"10.30.30.3","user":"cisco","password":"cisco"}

]

The result will be a Python list (the list of devices) of dictionaries (the specific device information for each device).

If instead the data in the file for the device info was defined as an array, as follows:

[

["ios-01","ios","10.30.30.1","cisco","cisco"],

["ios-02","ios","10.30.30.2","cisco","cisco"],

["ios-03","ios","10.30.30.3","cisco","cisco"]

]

The result would be a Python list (the list of devices) of lists (the lists of device information for each device).

Writing JSON data is similar but in reverse:

json\_device\_data = json.dumps(devices\_out\_list)

with open(json\_devices\_file, 'w') as json\_file:

json\_file.write(json\_device\_data)

Reading and writing is very simple with Python, as the translations from one to the other are direct and straightforward. JSON has many advantages and is quite popular, and therefore being able to read and write JSON data is an important skill in developing Python applications.

# 15.4 Binary Files

There may be times when textual data is not adequate for your needs, either because of limitations in functionality, or for performance reasons. Therefore, it may be useful to understand how to read and write binary data using Python.

**Opening**

In reading normal text files, you have been opening your files as follows:

file = open('myfile','r')

The second argument that is passed into the open function is the 'mode', which include 'r' for read and 'w' for write. By default, the file is assumed to be textual. However, if you want to open the file for binary reading or writing, you need to place a 'b' after the 'r' or 'w', which will tell Python that you are reading or writing a binary file.

To open a binary file for reading, the command would be:

file = open('bin\_file','rb')

And to open a binary file for writing, the command would be:

file = open('bin\_out\_file','wb')

**Reading and Writing**

You will use the same generic **read()** and **write()** functions for writing binary data. Python takes care of the details for you.

bin\_data = file.read()

file.write(bin\_data)

Consider an example where, for every device, you must store the [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH&type=terms) key or some other token for accessing devices programmatically. After reading in the general information about each device, in textual format, the binary data must be read for each device, and attached to the device object. The following code will read the appropriate binary file and attach the binary data to the device:

# Open SSH key file for this device

key\_file\_path = "sshkeys/"+device\_info['ip']+"/"+device\_info['key']

key\_file = open(key\_file\_path,'rb')

key\_data = key\_file.read() # read ssh key data

device.set\_sshkey(key\_data) # store ssh key data in device object

The first two lines of code create the path to the key using the device[IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms), and then open the file, using the mode **rb** for read-binary.

The third line reads the binary data into the variable key\_data, and the fourth line references the device object, calling a method 'set\_sshkey', passing the key\_data, so that the SSH key data can be attached to the device object, for later access purposes.

**Closing**

It is best practices to close any files that you are finished working on. This ensures that any resources that are taken up by the open file are freed up. The command to do this is the **close()**.

key\_file = open("info.txt", "r")

print key\_file.read()

key\_file.close()

# 15.5 Reading and Writing Structured Files

In this lab, you will read and write [CSV](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CSV&type=terms), [JSON](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=JSON&type=terms), and binary files.

The lab is ready!

Launch

## Reading and Writing CSV Files

In this exercise, you will:

* Read device information data from a CSV formatted file
* Create device objects with information for each device in the file
* Attempt to connect to the devices
* Print the results
* Output a CSV formatted file with information about each device from the list.

**Step 1**

Read in the CSV device information data from the csv-devices file located in the PRNE/section15/csv folder. Create device objects based on the device type of each device in the file.

### **Answer**

NOTE: The solution steps show code for each file denoted in parentheses. The file names are not part of the solution. The complete solution is posted in the PRNE/section15 folder.

|  |
| --- |
| (in main.py)  devices\_list = read\_devices\_info('csv-devices')  (in util.py)  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  file = open(devices\_file,'r') # Open the CSV file  csv\_devices = csv.reader(file) # Create the CSV reader for file  # Use list comprehension to put CSV data into list of lists  device\_info\_list = [dev\_info for dev\_info in csv\_devices]  for device\_info in device\_info\_list:  # Create a device object with this data  if device\_info[1] == 'ios':    device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  elif device\_info[1] == 'ios-xr':    device = NetworkDeviceXR(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device) # Append this device object to list  return devices\_list |

**Step 2**

Attempt to connect to, and retrieve device information from each device.

### **Answer**

|  |
| --- |
| (in main.py)  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  (in devclass.py)  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before |

**Step 3**

Print the results of every device connection attempt.

### **Answer**

|  |
| --- |
| (in main.py)  print\_device\_info(device) # print device details for this device  (in util.py)  #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print '-------------------------------------------------------' |

**Step 4**

Write the device information into a CSV file for every device.

### **Answer**

|  |
| --- |
| (in main.py)  write\_devices\_info('csv-devices-out', devices\_list)  (in util.py)  #====================================================================  def write\_devices\_info(devices\_file, devices\_list):  print '---- Printing CSV output ------------------------------'  # Create the list of lists with devices and device info  devices\_out\_list = [] # create list for CSV output  for device in devices\_list:  dev\_info = [device.name,device.ip\_address,device.interfaces != ""]  devices\_out\_list.append(dev\_info)    pprint(devices\_out\_list)  # Use CSV library to output our list of lists to a CSV file  with open(devices\_file, 'w') as file:  csv\_out = csv.writer(file)  csv\_out.writerows(devices\_out\_list) |

**Step 5**

Run your application and verify the CSV file is created.

### **Answer**

Following is the complete application.

|  |
| --- |
| **File: util.py**  import csv  from pprint import pprint  from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  from devclass import NetworkDeviceXR  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  file = open(devices\_file,'r') # Open the CSV file  csv\_devices = csv.reader(file) # Create the CSV reader for file  # Iterate through all devices in our CSV file  for device\_info in csv\_devices:  # Create a device object with this data  if device\_info[1] == 'ios':    device = NetworkDeviceIOS(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  elif device\_info[1] == 'ios-xr':    device = NetworkDeviceXR(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  else:  device = NetworkDevice(device\_info[0],device\_info[2],  device\_info[3],device\_info[4])  devices\_list.append(device) # Append this device object to list  return devices\_list    #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print '-------------------------------------------------------'  #====================================================================  def write\_devices\_info(devices\_file, devices\_list):  print '---- Printing CSV output ------------------------------'  # Create the list of lists with devices and device info  devices\_out\_list = [] # create list for CSV output  for device in devices\_list:  dev\_info = [device.name,device.ip\_address,device.interfaces != ""]  devices\_out\_list.append(dev\_info)    pprint(devices\_out\_list)  # Use CSV library to output our list of lists to a CSV file  with open(devices\_file, 'w') as file:  csv\_out = csv.writer(file)  csv\_out.writerows(devices\_out\_list)    **File: devclass.py**  import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before    #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting XR: ssh '+self.username+'@'+self.ip\_address  self.session = pexpect.spawn('ssh '+self.username+  '@'+self.ip\_address, timeout=20)  result = self.session.expect(['password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timout or unexpected reply from device'  return 0  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  self.session.expect('#')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before    **File: main.py**  from util import read\_devices\_info  from util import print\_device\_info  from util import write\_devices\_info  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('csv-devices') # read CSV info for all devices  for device in devices\_list:  print '==== Device ============================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device  write\_devices\_info('csv-devices-out', devices\_list) # write CSV entry for all devices |

## Reading and Writing JSON Files

In this exercise, you will:

* Read device information data from a JSON formatted file
* Create device objects with information for each device in the file
* Attempt to connect to the devices
* Output a JSON formatted file with information about each device from the list.

You will then read in the new JSON file and connect to the devices again to verify that the writing of data was successful.

**Step 6**

Read device information from the file **json-devices** that is located in the PRNE/section15/json folder. Create device objects for each device in the file.

### **Answer**

|  |
| --- |
| (in main.py)  devices\_list = read\_devices\_info('json-devices') # read JSON info for all devices  (in util.py)  import json  from pprint import pprint  from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  from devclass import NetworkDeviceXR  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  # Open the device file with JSON data and read into string  json\_file = open(devices\_file,'r') # open the JSON file  json\_device\_data = json\_file.read() # read in the JSON data from file  # Convert JSON string into Python data structure  devices\_info\_list = json.loads(json\_device\_data)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':    device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  devices\_list.append(device) # Append this device object to list  return devices\_list |

**Step 7**

For every device, connect using Pexpect and get interface data.

### **Answer**

|  |
| --- |
| (main.py)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':    device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  devices\_list.append(device) # Append this device object to list  return devices\_list  (devclass.py)  #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios-xr'  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting XR: ssh '+self.username+'@'+self.ip\_address  self.session = pexpect.spawn('ssh '+self.username+  '@'+self.ip\_address, timeout=20)  result = self.session.expect(['password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timout or unexpected reply from device'  return 0  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  self.session.expect('#')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before |

**Step 8**

Print the device information for every device.

### **Answer**

|  |
| --- |
| (main.py)  for device in devices\_list:  print '==== Device ============================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device  (util.py)  #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print '-------------------------------------------------------' |

**Step 9**

Write device information to a JSON formatted file, which will hold the information about each device from your device objects.

### **Answer**

|  |
| --- |
| (main.py)  write\_devices\_info('json-devices-out', devices\_list) # write JSON entry for all devices  (util.py)  #====================================================================  def write\_devices\_info(devices\_file, devices\_list):  print '---- Printing JSON output ------------------------------'  # Create the list of lists with devices and device info  devices\_out\_list = [] # create list for JSON output  for device in devices\_list:  dev\_info = {'name':device.name,'ip':device.ip\_address,'os':device.os\_type,  'user':device.username,'password':device.password}  devices\_out\_list.append(dev\_info)    pprint(devices\_out\_list)  # Convert the python device data into JSON for output to file  json\_device\_data = json.dumps(devices\_out\_list)  # Output the JSON string to a file  with open(devices\_file, 'w') as json\_file:  json\_file.write(json\_device\_data) |

**Step 10**

Read in the new file, creating device objects for each device.

### **Answer**

|  |
| --- |
| # Do it again, reading from your output file, to prove we did it correctly  print '-------------------------------------------------------------------'  print '---------- Reading from our output file, doing it again -----------'  devices\_list = read\_devices\_info('json-devices-out') # read JSON info for all devices |

**Step 11**

For every device, connect using Pexpect and get interface data. Print the device information.

### **Answer**

|  |
| --- |
| for device in devices\_list:  print '==== Device =================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device |

**Step 12**

Run your application and verify that the JSON output file was created.

### **Answer**

|  |
| --- |
| **File: util.py**  import json  from pprint import pprint  from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  from devclass import NetworkDeviceXR  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  # Open the device file with JSON data and read into string  json\_file = open(devices\_file,'r') # open the JSON file  json\_device\_data = json\_file.read() # read in the JSON data from file  # Convert JSAON string into Python data structure  devices\_info\_list = json.loads(json\_device\_data)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':    device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  devices\_list.append(device) # Append this device object to list  return devices\_list    #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print '-------------------------------------------------------'  #====================================================================  def write\_devices\_info(devices\_file, devices\_list):  print '---- Printing JSON output ------------------------------'  # Create the list of lists with devices and device info  devices\_out\_list = [] # create list for JSON output  for device in devices\_list:  dev\_info = {'name':device.name,'ip':device.ip\_address,'os':device.os\_type,  'user':device.username,'password':device.password}  devices\_out\_list.append(dev\_info)    pprint(devices\_out\_list)  # Convert the python device data into JSON for output to file  json\_device\_data = json.dumps(devices\_out\_list)  # Output the JSON string to a file  with open(devices\_file, 'w') as json\_file:  json\_file.write(json\_device\_data)    **File: devclass.py**  import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  self.os\_type = None  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios'  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before    #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios-xr'  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting XR: ssh '+self.username+'@'+self.ip\_address  self.session = pexpect.spawn('ssh '+self.username+  '@'+self.ip\_address, timeout=20)  result = self.session.expect(['password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timout or unexpected reply from device'  return 0  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  self.session.expect('#')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before    **File: main.py**  from util import read\_devices\_info  from util import print\_device\_info  from util import write\_devices\_info  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('json-devices') # read JSON info for all devices  for device in devices\_list:  print '==== Device ============================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device  write\_devices\_info('json-devices-out', devices\_list) # write JSON entry for all devices  # Do it again, reading from our output file, to prove we did it correctly  print '-----------------------------------------------------------------------------'  print '---------- Reading from our output file, doing it again ---------------------'  devices\_list = read\_devices\_info('json-devices-out') # read JSON info for all devices  for device in devices\_list:  print '==== Device ============================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device |

## Reading Binary Files

In this exercise, you will read information about network devices, and then for each device will read a binary file associated with that device representing its [SSH](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SSH&type=terms) key.

For the purposes of this lab exercise, the contents of the binary file are not relevant – the exercise is only covering the operation of reading binary data and storing it in Python objects. This exercise is simulating binary SSH key files, one for each network device.

You will then read in the new JSON file and connect to the devices again to verify that the writing of data was successful.

**Step 13**

Read device information from the **json-devices** file located in the PRNE/section15/binary folder. Create device objects containing the device name, IP address, username, and password.

### **Answer**

NOTE: Only the new code is shown for each step. The entire solution is available at the end of the exercise.

|  |
| --- |
| #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  # Open the device file with JSON data and read into string  json\_file = open(devices\_file,'r') # open the JSON file  json\_device\_data = json\_file.read() # read in the JSON data from file  # Convert JSAON string into Python data structure  devices\_info\_list = json.loads(json\_device\_data)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':    device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password']) |

**Step 14**

For each device, use the device[IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms) to read SSH key data (just a binary file) located in the PRNE/section15/binary/ sshkeys/<device-IP-address>/<keyname> folder.

### **Answer**

|  |
| --- |
| # Open SSH key file for this device  key\_file\_path = "sshkeys/"+device\_info['ip']+"/"+device\_info['key']  key\_file = open(key\_file\_path,'rb')  key\_data = key\_file.read() # read ssh key data |

**Step 15**

After reading the file, pass the binary data object to the device object. The device object will need to implement a method such as 'set\_sshkey' which is passed the binary data object you have just read.

### **Answer**

|  |
| --- |
| device.set\_sshkey(key\_data) # store ssh key data in device object |

**Step 16**

When complete, print the information for each device, including the binary key data file. Print the key data in hexadecimal.

Note: printing hexadecimal values for your binary data can be accomplished by using a for loop, iterating through every character in the 'string' (which is how the binary data is stored), and using the 'encode()' function to convert each character to hex for output.

|  |
| --- |
| for c in device.sshkey: print c.encode('hex'), |

### **Answer**

|  |
| --- |
| #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print ' Device key: ',  for c in device.sshkey: print c.encode('hex'),  print '\n-------------------------------------------------------' |

**Step 17**

Run your application and verify that the SSH keys are printed.

### **Answer**

The entire application should look similar to:

|  |
| --- |
| **File: util.py**  import json  from pprint import pprint  from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  from devclass import NetworkDeviceXR  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  # Open the device file with JSON data and read into string  json\_file = open(devices\_file,'r') # open the JSON file  json\_device\_data = json\_file.read() # read in the JSON data from file  # Convert JSAON string into Python data structure  devices\_info\_list = json.loads(json\_device\_data)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':    device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  # Open SSH key file for this device  key\_file\_path = "sshkeys/"+device\_info['ip']+"/"+device\_info['key']  key\_file = open(key\_file\_path,'rb')  key\_data = key\_file.read() # read ssh key data  device.set\_sshkey(key\_data) # store ssh key data in device object  devices\_list.append(device) # Append this device object to list  return devices\_list    #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print ' Device key: ',  for c in device.sshkey: print c.encode('hex'),  print '\n-------------------------------------------------------'  #====================================================================  def write\_devices\_info(devices\_file, devices\_list):  print '---- Printing JSON output ------------------------------'  # Create the list of lists with devices and device info  devices\_out\_list = [] # create list for JSON output  for device in devices\_list:  dev\_info = {'name':device.name,'ip':device.ip\_address,'os':device.os\_type,  'user':device.username,'password':device.password}  devices\_out\_list.append(dev\_info)    pprint(devices\_out\_list)  # Convert the python device data into JSON for output to file  json\_device\_data = json.dumps(devices\_out\_list)  # Output the JSON string to a file  with open(devices\_file, 'w') as json\_file:  json\_file.write(json\_device\_data)    **File: devclass.py**  import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  self.os\_type = None  self.sshkey = None  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  # ---- Set SSH Key ------------------------------------------------  def set\_sshkey(self, sshkey):  self.sshkey = sshkey  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios'  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before    #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios-xr'  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting XR: ssh '+self.username+'@'+self.ip\_address  self.session = pexpect.spawn('ssh '+self.username+  '@'+self.ip\_address, timeout=20)  result = self.session.expect(['password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timout or unexpected reply from device'  return 0  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  self.session.expect('#')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before    **File: main.py**  from util import read\_devices\_info  from util import print\_device\_info  from util import write\_devices\_info  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('json-devices') # read JSON info for all devices  for device in devices\_list:  print '==== Device ============================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device |

# 15.6 Using Databases for Storing Networking Data

There are several aspects of database communication which must be understood in order to make use of the functionality afforded by this type of storage. In particular:

* **Connection**: Before your application can communicate with the database, you must establish a connection to the [DB](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=DB&type=terms). For sqlite3, the connection is established using the 'connect()' function.
* **Cursor**: Once a connection has been established, your communication with the database will be using the cursor, which can be thought of as a pointer to a row in a database table. The cursor tells the [SQL](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=SQL&type=terms) library the table and row upon which an operation is to be performed.
* **SQL Commands**: Your application will interact with the database using Structured Query Language (SQL). SQL commands operate on tables and rows in the database, and some common commands allow you to create tables (CREATE), add or replace rows (REPLACE), and search for specific items (SEARCH).
* **sqlite3**: SQLite is a lightweight library for accessing SQL databases, and is well-suited to modest-sized applications and for prototyping solutions. SQLite provides simple functions for connecting to the database, establishing the cursor, and executing SQL commands.

###### Creating Tables

One of the basic tasks when dealing with databases is the creation of a table. The process begins with establishing the connection and getting the cursor for the database.

# Connect to the database and get the cursor

db\_connection = sqlite3.connect(devices\_db\_file) # DB connection

db\_cursor = db\_connection.cursor() # DB cursor

The next step is to issue the SQL command to create a table. The SQL "CREATE TABLE" command is used for this purpose, providing the name of the table, and the columns representing the data items of the table.

In the following example, the code is executing an SQL command to create a table called "devices," creating columns (fields) for device name, [IP address,](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms) [OS](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=OS&type=terms)-type, username, and password.

db\_cursor.execute('''CREATE TABLE IF NOT EXISTS devices

(name VARCHAR(16) PRIMARY KEY,

ip VARCHAR(16),

os VARCHAR(8),

user VARCHAR(16),

pw VARCHAR(16))''')

In the SQL command above:

* **db\_cursor.execute():**The SQLite function that is called to execute an SQL command.
* **SQL command**: "CREATE TABLE IF NOT EXISTS devices" will create a table in the database by the given name ('devices'), unless it exists.
* **name, ip, os, user, pw:** All the fields for the table, the specific items of information about each device that will be stored.
* **VARCHAR:** The 'type' of data that will be stored – 'VARCHAR' is the SQL name for a character string of variable length. The value in parentheses, for example (16), specifies the maximum length.
* **PRIMARY KEY:** Specifies the key for the table. Primary keys must be unique within the table and may be used to improve efficiency and insure deterministic behavior of the database table.

###### Writing Device Information to a Table

Writing data to an existing database table can be done using the SQL REPLACE command, which will either add or replace the item based on the prior existence of the given primary key value.

In Python the procedure is to use SQLite and your cursor object to execute an SQL command to write the desired data into the database table. The first step is to connect to the database:

# Connect to the database and get a connection to the database

db\_connection = sqlite3.connect(devices\_db\_file) # DB connection

db\_cursor = db\_connection.cursor() # DB cursor

Next, create the SQL command.

sql\_cmd = 'REPLACE INTO devices (name, ip, os, user, pw) VALUES(?,?,?,?,?)'

db\_cursor.execute(sql\_cmd, (device.name,

device.ip\_address,

device.os\_type,

device.username,

device.password))

In the code above:

* **SQL Command**: "REPLACE INTO devices" tells SQLite that the application is intending to add or replace a row in the devices table.
* **(name,ip,os,user,pw):** The fields into which the data will be placed.
* **VALUES(?,?,?,?,?):** Placeholders for the values that will be provided in the SQLite execute command, making it easy to write applications that pass in the appropriate data to be filled into the SQL command when it is actually executed by the database.
* **db\_cursor.execute():** The last step is to pass this command in to SQLite, telling it to execute the command, and passing in the variables that will be substituted for the question marks (?,?,?,?,?) in the actual command.

###### Reading Device Information from a Table

Gathering information from a database is a matter of performing a 'SELECT' operation and having the result delivered into your Python data structures.

First, connect to the database:

# Connect to the database and get a connection to the database

db\_connection = sqlite3.connect(devices\_db\_file) # DB connection

db\_cursor = db\_connection.cursor() # DB cursor

Next, execute the SELECT command, specifying to get all rows ('\*') from the table 'devices':

db\_cursor.execute('SELECT \* FROM devices')

devices\_from\_db = db\_cursor.fetchall()

The resulting Python object will be a list, of tuples:

* The outer list is a list of the rows of the table – in this case, one item for each device.
* The inner tuple is a sequence of the device information, one value for each field in the table.

The resulting data would look similar to:

[('ios-01', '10.30.30.1', 'ios', 'cisco', 'cisco'),

('ios-02', '10.30.30.2', 'ios', 'cisco', 'cisco'),

('ios-03', '10.30.30.3', 'ios', 'cisco', 'cisco')]

# 15.7 Reading and Writing Database Files

In this lab, you will:

* Read device information from a[JSON](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=JSON&type=terms) file
* Create device objects for the network devices in your lab environment
* Connect to each device
* Get interface information
* Write the device information into a database file

You will then read the information from the database file you have just created, in order to insure that the data has been written correctly.

Lab is currently initializing: 26% complete

## Reading and Writing Database Files

**Step 1**

Read device information from a JSON file, and create a list of network device objects holding connectivity information for each device.

### **Answer**

|  |
| --- |
| #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  # Open the device file with JSON data and read into string  json\_file = open(devices\_file,'r') # open the JSON file  json\_device\_data = json\_file.read() # read in the JSON data from file  # Convert JSON string into Python data structure  devices\_info\_list = json.loads(json\_device\_data)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':  device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  devices\_list.append(device) # Append this device object to list  return devices\_list |

**Step 2**

Connect to each device and get interface information, printing device information at each iteration.

### **Answer**

|  |
| --- |
| #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios'  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before |

**Step 3**

Use the sqlite3 library to write out device information into a database file.

### **Answer**

|  |
| --- |
| #====================================================================  def write\_devices\_db(devices\_db\_file, devices\_list):  print '---- writing devices to db ------------------------------'  # Connect to the database and get a connection  db\_connection = sqlite3.connect(devices\_db\_file) # DB connection  db\_cursor = db\_connection.cursor() # DB cursor  # Create our devices table in the database we just opened  db\_cursor.execute('''CREATE TABLE IF NOT EXISTS devices  (name VARCHAR(16) PRIMARY KEY,  ip VARCHAR(16),  os VARCHAR(8),  user VARCHAR(16),  pw VARCHAR(16))''')  # Iterate through devices, printing one per line in devices table  for device in devices\_list:    # Insert or replace the device information into devices table  sql\_cmd = 'REPLACE INTO devices (name,ip,os,user,pw) VALUES(?,?,?,?,?)'  db\_cursor.execute(sql\_cmd, (device.name,  device.ip\_address,  device.os\_type,  device.username,  device.password))  db\_connection.commit() # Must commit changes or they are not saved!  db\_cursor.close()  db\_connection.close() |

**Step 4**

Use the sqlite3 library to read information from the database file.

### **Answer**

|  |
| --- |
| #=====================================================================  def read\_devices\_db(devices\_db\_file):  print '---- reading devices from db ----------------------------'  # Connect to the database and get a connection  db\_connection = sqlite3.connect(devices\_db\_file) # DB connection  db\_cursor = db\_connection.cursor() # DB cursor  db\_cursor.execute('SELECT \* FROM devices')  devices\_from\_db = db\_cursor.fetchall()  pprint(devices\_from\_db)  print '---- end devices from db --------------------------------'  # Done reading devices from the table, close it down.  db\_cursor.close()  db\_connection.close()    return devices\_from\_db |

**Step 5**

Print the device information that has just been read from the database.

### **Answer**

|  |
| --- |
| pprint(devices\_from\_db) |

**Step 6**

Run your application and verify that the data was read from the database.

### **Answer**

Your complete application should look similar to:

|  |
| --- |
| **file: util.py**  import sqlite3  import json  from pprint import pprint  from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  from devclass import NetworkDeviceXR  #======================================================================  def read\_devices\_info(devices\_file):  devices\_list = []  # Open the device file with JSON data and read into string  json\_file = open(devices\_file,'r') # open the JSON file  json\_device\_data = json\_file.read() # read in the JSON data from file  # Convert JSAON string into Python data structure  devices\_info\_list = json.loads(json\_device\_data)  for device\_info in devices\_info\_list:  # Create a device object with this data  if device\_info['os'] == 'ios':    device = NetworkDeviceIOS(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  elif device\_info['os'] == 'ios-xr':    device = NetworkDeviceXR(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  else:  device = NetworkDevice(device\_info['name'],device\_info['ip'],  device\_info['user'],device\_info['password'])  devices\_list.append(device) # Append this device object to list  return devices\_list    #====================================================================  def print\_device\_info(device):  print '-------------------------------------------------------'  print ' Device Name: ',device.name  print ' Device IP: ',device.ip\_address  print ' Device username: ',device.username,  print ' Device password: ',device.password  print '-------------------------------------------------------'  #====================================================================  def write\_devices\_db(devices\_db\_file, devices\_list):  print '---- writing devices to db ------------------------------'  # Connect to the database and get a connection  db\_connection = sqlite3.connect(devices\_db\_file) # DB connection  db\_cursor = db\_connection.cursor() # DB cursor  # Create our devices table in the database we just opened  db\_cursor.execute('''CREATE TABLE IF NOT EXISTS devices  (name VARCHAR(16) PRIMARY KEY,  ip VARCHAR(16),  os VARCHAR(8),  user VARCHAR(16),  pw VARCHAR(16))''')  # Iterate through devices, printing one per line in devices table  for device in devices\_list:    # Insert or replace the device information into devices table  sql\_cmd = 'REPLACE INTO devices (name, ip, os, user, pw) VALUES(?,?,?,?,?)'  db\_cursor.execute(sql\_cmd, (device.name,  device.ip\_address,  device.os\_type,  device.username,  device.password))  db\_connection.commit() # Must commit changes or they are not saved!  db\_cursor.close()  db\_connection.close()    #=====================================================================  def read\_devices\_db(devices\_db\_file):  print '---- reading devices from db ----------------------------'  # Connect to the database and get a connection  db\_connection = sqlite3.connect(devices\_db\_file) # DB connection  db\_cursor = db\_connection.cursor() # DB cursor  db\_cursor.execute('SELECT \* FROM devices')  devices\_from\_db = db\_cursor.fetchall()  pprint(devices\_from\_db)  print '---- end devices from db --------------------------------'  # Done reading devices from the table, close it down.  db\_cursor.close()  db\_connection.close()    return devices\_from\_db    **file: devclass.py**  import pexpect  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  self.os\_type = None  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios'  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    def get\_interfaces(self):    self.session.sendline('show interfaces summary')  result = self.session.expect('>')  self.interfaces = self.session.before    #---- Class to hold information about an IOS-XR network device --------  class NetworkDeviceXR(NetworkDevice):  #---- Initialize --------------------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios-xr'  #---- Connect to device -------------------------------------------  def connect(self):  print '--- connecting XR: ssh '+self.username+'@'+self.ip\_address  self.session = pexpect.spawn('ssh '+self.username+  '@'+self.ip\_address, timeout=20)  result = self.session.expect(['password:', pexpect.TIMEOUT])  # Check for failure  if result != 0:  print '--- Timout or unexpected reply from device'  return 0  # Successfully got password prompt, logging in with password  print '--- password:',self.password  self.session.sendline(self.password)  self.session.expect('#')  #---- Get interfaces from device ----------------------------------  def get\_interfaces(self):  self.session.sendline('show interface brief')  result = self.session.expect('#')  self.interfaces = self.session.before    **file: main.py**  from util import read\_devices\_info  from util import print\_device\_info  from util import write\_devices\_db  from util import read\_devices\_db  from pprint import pprint  #====================================================================  # Main program: connect to device, show interface, display  devices\_list = read\_devices\_info('json-devices') # read JSON info for all devices  for device in devices\_list:  print '==== Device ============================================================='  device.connect() # connect to this specific device  device.get\_interfaces() # get interface info for this specific device  print\_device\_info(device) # print device details for this device  devices\_db\_file = 'devices.db'  write\_devices\_db(devices\_db\_file, devices\_list) # write device data to database  # Now read in the device information we just wrote  devices\_from\_db = read\_devices\_db(devices\_db\_file)  print '==== Device info from database ============================================='  pprint(devices\_from\_db) |

# 15.8 Storing Traffic Data

In this lab, you will create an application that gathers traffic data from an interface on a device in the network. Your application will poll for this traffic data a specific number of times, at a specific interval. When complete, it will print [LAN](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=LAN&type=terms)utilization statistics for the chosen interface.

Your application will:

* Receive interval and count information from the command line, as well as the target device and interface.
* Read device info from the database
* Connect to device and gather the appropriate number of traffic statistics readings from the device
* Print the results when complete

**Step 1**

Allow the user to enter command-line arguments to set the interval, count, device [IP address](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IP%20address&type=terms), and interface name **gigabitethernet 0/1**. The command line should look like:

|  |
| --- |
| python main.py [interval] [count] [device IP] [interface] |

There should be default values for each variable. In Python, command-line arguments are accessed using the variable **argv**, where **argv[0]** is the application filename (for example, 'main.py'), **argv[1]** is the first parameter, **argv[2]** is the second, and so on. You will need to import **argv** from **sys**.

### **Answer**

The **argv** list, available once you import it from **sys**, is a list with each argument presented to the script on the command line.

|  |
| --- |
| from sys import argv  #====================================================================  # Main program: read devices, then get traffic statistics from each  # Get arguments passed to our application  print 'argv: ',argv  interval = int(argv[1]) if len(argv) >= 2 else 5  count = int(argv[2]) if len(argv) >= 3 else 5  device\_ip = argv[3] if len(argv) >= 4 else '10.30.30.1'  interface = argv[4] if len(argv) >= 5 else 'gigabitethernet 0/1' |

**Step 2**

Read in your device information from the database **devices.db** located in the **~/Desktop/PRNE/section15/db**folder.

### **Answer**

|  |
| --- |
| #======================================================================  def read\_devices\_db(devices\_db\_file):  print '---- reading devices from db ----------------------------'  # Connect to the database and get a connection  db\_connection = sqlite3.connect(devices\_db\_file) # DB connection  db\_cursor = db\_connection.cursor() # DB cursor  db\_cursor.execute('SELECT \* FROM devices')  devices\_from\_db = db\_cursor.fetchall()  pprint(devices\_from\_db)  print '---- end devices from db --------------------------------'  # Done adding devices to the table, close it down.  db\_cursor.close()  db\_connection.close()    return devices\_from\_db |

**Step 3**

Find the device access information (username and password) for the target device from the information read from the database.

### **Answer**

|  |
| --- |
| #======================================================================  def get\_device(devices\_from\_db, device\_ip):  for device\_info in devices\_from\_db:  # Create a device object with this data  if device\_info[1] == device\_ip and device\_info[2] == 'ios':    return NetworkDeviceIOS(device\_info[0],device\_info[1],  device\_info[3],device\_info[4])  return None |

**Step 4**

Gather traffic data from the specified interface for the target device, for the appropriate interval and count. Write the information to a [CSV](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CSV&type=terms) log file.

### **Answer**

|  |
| --- |
| #=======================================================================  def gather\_traffic\_data(logfile, device, interface, interval, count):  # Reset the log file to begin fresh  dev\_stats\_log = open(logfile,'w')  dev\_stats\_log.close()  # Loop for 'num\_readings' number of times  for \_ in range(count): # Loop the specified number of times  print ''  print '---- reading rx, tx data ------------------------------'  dev\_stats\_log = open(logfile,'a') # open the file for appending  # Get the rx and tx stats for a specific interface on device  device.connect()  stats = device.get\_interface\_stats(interface)  print ' ---- ',device.ip\_address,' : stats: ',stats  # Write rx and tx information to database  write\_stats\_log(dev\_stats\_log, time(), device.ip\_address,  stats[0], stats[1], stats[2], stats[3])  dev\_stats\_log.close() # Done appending to file for now  print ''  sleep(interval) # Pause the defined interval |

**Step 5**

At each iteration, write the log entries to the log file.

### **Answer**

|  |
| --- |
| #======================================================================  def write\_stats\_log(file, time, ip\_address,  rx\_packets, rx\_bytes,  tx\_packets, tx\_bytes):  # Create a log entry from the data given  log\_entry = [time, ip\_address, rx\_packets, rx\_bytes,  tx\_packets, tx\_bytes]  log = csv.writer(file)  log.writerow(log\_entry) |

**Step 6**

After the gathering interval has completed, read in the log file, and calculate and print the LAN utilization for each interval.

### **Answer**

|  |
| --- |
| #======================================================================  def print\_log(device\_ip\_address, log\_info\_list):  print ' Time Rx Packets Rx Bytes Tx Packets Tx Bytes Util'  print ' ---------- ---------- ---------- ---------- ---------- -------'  last\_bytes = 0  for log\_entry in log\_info\_list:  if log\_entry[1] != device\_ip\_address: # Ignore if not passed IP address  continue  if last\_bytes == 0: # if first calculation, set util to 0  util = 0  last\_bytes = int(log\_entry[3]) + int(log\_entry[5])  last\_secs = float(log\_entry[0])  else: # else calculate lan utilization  current\_bytes = int(log\_entry[3]) + int(log\_entry[5])  interval\_bytes = current\_bytes - last\_bytes    if\_speed = 1000000000  current\_secs = float(log\_entry[0])  interval\_time = current\_secs - last\_secs  util = (interval\_bytes\*8\*100) / float(interval\_time\*if\_speed)  last\_bytes = current\_bytes  last\_secs = current\_secs  str\_time = strftime('%H:%M:%S', localtime(float(log\_entry[0])))  print ' {0:10} {1:>10} {2:>10} {3:>10} {4:>10} {5:>7.3f}'.format(str\_time,  log\_entry[2], log\_entry[3],  log\_entry[4], log\_entry[5],  util) |

**Step 7**

Run your application and verify the output.

### **Answer**

Your output should look similar to:

|  |
| --- |
| $ python main.py 2 5  argv: ['main.py', '2', '5']  ---- reading devices from db ----------------------------  [(u'ios-01', u'10.30.30.1', u'ios', u'cisco', u'cisco'),  (u'ios-02', u'10.30.30.2', u'ios', u'cisco', u'cisco'),  (u'ios-03', u'10.30.30.3', u'ios', u'cisco', u'cisco')]  ---- end devices from db --------------------------------  ---- reading rx, tx data ------------------------------  --- connecting IOS: telnet 10.30.30.1  ---- 10.30.30.1 : stats: ('88429', '84210375', '91849', '85053465')  ---- reading rx, tx data ------------------------------  --- connecting IOS: telnet 10.30.30.1  ---- 10.30.30.1 : stats: ('88575', '84325202', '92015', '85170709')  ---- reading rx, tx data ------------------------------  --- connecting IOS: telnet 10.30.30.1  ---- 10.30.30.1 : stats: ('88713', '84427781', '92173', '85275699')  ---- reading rx, tx data ------------------------------  --- connecting IOS: telnet 10.30.30.1  ---- 10.30.30.1 : stats: ('88849', '84530234', '92330', '85380629')  ---- reading rx, tx data ------------------------------  --- connecting IOS: telnet 10.30.30.1  ---- 10.30.30.1 : stats: ('88989', '84632927', '92487', '85485559')  Device: 10.30.30.1 Interface: gigabitethernet 0/1  Time Rx Packets Rx Bytes Tx Packets Tx Bytes Util  ---------- ---------- ---------- ---------- ---------- -------  11:11:39 88429 84210375 91849 85053465 0.000  11:11:42 88575 84325202 92015 85170709 0.069  11:11:45 88713 84427781 92173 85275699 0.061  11:11:48 88849 84530234 92330 85380629 0.060  11:11:50 88989 84632927 92487 85485559 0.060 |

You complete application utility library should look similar to:

|  |
| --- |
| **#file: util.py**  import csv  from pprint import pprint  import sqlite3  from time import strftime  from time import localtime  from time import sleep  from time import time  from devclass import NetworkDevice  from devclass import NetworkDeviceIOS  #======================================================================  def read\_devices\_db(devices\_db\_file):  print '---- reading devices from db ----------------------------'  # Connect to the database and get a connection  db\_connection = sqlite3.connect(devices\_db\_file) # DB connection  db\_cursor = db\_connection.cursor() # DB cursor  db\_cursor.execute('SELECT \* FROM devices')  devices\_from\_db = db\_cursor.fetchall()  pprint(devices\_from\_db)  print '---- end devices from db --------------------------------'  # Done adding devices to the table, close it down.  db\_cursor.close()  db\_connection.close()    return devices\_from\_db  #======================================================================  def get\_device(devices\_from\_db, device\_ip):  for device\_info in devices\_from\_db:  # Create a device object with this data  if device\_info[1] == device\_ip and device\_info[2] == 'ios':    return NetworkDeviceIOS(device\_info[0],device\_info[1],  device\_info[3],device\_info[4])  return None    #=======================================================================  def gather\_traffic\_data(logfile, device, interface, interval, count):  # Reset the log file to begin fresh  dev\_stats\_log = open(logfile,'w')  dev\_stats\_log.close()  # Loop for 'num\_readings' number of times  for \_ in range(count): # Loop the specified number of times  print ''  print '---- reading rx, tx data ------------------------------'  dev\_stats\_log = open(logfile,'a') # open the file for appending  # Get the rx and tx stats for a specific interface on device  device.connect()  stats = device.get\_interface\_stats(interface)  print ' ---- ',device.ip\_address,' : stats: ',stats  # Write rx and tx information to database  write\_stats\_log(dev\_stats\_log, time(), device.ip\_address,  stats[0], stats[1], stats[2], stats[3])  dev\_stats\_log.close() # Done appending to file for now  print ''  sleep(interval) # Pause the defined interval  #======================================================================  def write\_stats\_log(file, time, ip\_address,  rx\_packets, rx\_bytes,  tx\_packets, tx\_bytes):  # Create a log entry from the data given  log\_entry = [time, ip\_address, rx\_packets, rx\_bytes,  tx\_packets, tx\_bytes]  log = csv.writer(file)  log.writerow(log\_entry)    #======================================================================  def print\_log(device\_ip\_address, log\_info\_list):  print ' Time Rx Packets Rx Bytes Tx Packets Tx Bytes Util'  print ' ---------- ---------- ---------- ---------- ---------- -------'  last\_bytes = 0  for log\_entry in log\_info\_list:  if log\_entry[1] != device\_ip\_address: # Ignore if not passed IP address  continue  if last\_bytes == 0: # if first calculation, set util to 0  util = 0  last\_bytes = int(log\_entry[3]) + int(log\_entry[5])  last\_secs = float(log\_entry[0])  else: # else calculate lan utilization  current\_bytes = int(log\_entry[3]) + int(log\_entry[5])  interval\_bytes = current\_bytes - last\_bytes    if\_speed = 1000000000  current\_secs = float(log\_entry[0])  interval\_time = current\_secs - last\_secs  util = (interval\_bytes\*8\*100) / float(interval\_time\*if\_speed)  last\_bytes = current\_bytes  last\_secs = current\_secs  str\_time = strftime('%H:%M:%S', localtime(float(log\_entry[0])))  print ' {0:10} {1:>10} {2:>10} {3:>10} {4:>10} {5:>7.3f}'.format(str\_time,  log\_entry[2], log\_entry[3],  log\_entry[4], log\_entry[5],  util) |

You complete application device library should look similar to:

|  |
| --- |
| **#file: devclass.py**  import pexpect  #===================================================================================  #---- Class to hold information about a generic network device --------  class NetworkDevice():  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  self.name = name  self.ip\_address = ip  self.username = user  self.password = pw  self.os\_type = None  def connect(self):  self.session = None  def get\_interfaces(self):  self.interfaces = '--- Base Device, does not know how to get interfaces ---'  def get\_interface\_stats(self,interface):  return (0,0,0,0)    #===================================================================================  #---- Class to hold information about an IOS network device --------  class NetworkDeviceIOS(NetworkDevice):  #---- Initialization ---------------------------------------  def \_\_init\_\_(self, name, ip, user='cisco', pw='cisco'):  NetworkDevice.\_\_init\_\_(self, name, ip, user, pw)  self.os\_type = 'ios'  #---- Connect -----------------------------------------------  def connect(self):  print '--- connecting IOS: telnet '+self.ip\_address  self.session = pexpect.spawn('telnet '+self.ip\_address, timeout=20)  result = self.session.expect(['Username:', pexpect.TIMEOUT])  self.session.sendline(self.username)  result = self.session.expect('Password:')  # Successfully got password prompt, logging in with password  self.session.sendline(self.password)  self.session.expect('>')    #---- Get Stats ---------------------------------------------  def get\_interface\_stats(self, interface):  stats\_cmd = 'show interface ' + interface + ' accounting' \  + ' | include IP'  # Execute the show interface accounting command  self.session.sendline(stats\_cmd)  result = self.session.expect('>')  stats\_output = (self.session.before).splitlines()  for stats\_line in stats\_output:  stats = stats\_line.split()  if stats[0] == 'IP': # We only care about 'IP' stats  return (stats[1],stats[2],stats[3],stats[4])  print '--- unexpected show interface output'  return (0,0,0,0) |

You complete application should look similar to:

|  |
| --- |
| **file: main.py**  from util import read\_devices\_db  from util import get\_device  from util import gather\_traffic\_data  from util import write\_stats\_log  from util import print\_log  from sys import argv  import csv  #====================================================================  # Main program: read devices, then get traffic statistics from each  # Get arguments passed to our application  print 'argv: ',argv  interval = int(argv[1]) if len(argv) >= 2 else 5  count = int(argv[2]) if len(argv) >= 3 else 5  device\_ip = argv[3] if len(argv) >= 4 else '10.30.30.1'  interface = argv[4] if len(argv) >= 5 else 'gigabitethernet 0/1'  # Read device information from database, into list of device info lists  devices\_from\_db = read\_devices\_db('devices.db')  # Get device information for our device  device = get\_device(devices\_from\_db, device\_ip)  if device == None:  print '!!! Cannot find device in DB!'  exit()  logfile = 'dev-stats-log' # set output CSV log file  # Gather traffic data for the devices in the list  gather\_traffic\_data(logfile, device, interface, interval, count)  dev\_stats\_log = open(logfile,'r')  csv\_log = csv.reader(dev\_stats\_log)  log\_info\_list = [log\_info for log\_info in csv\_log]  # Print log information for our one device  print ''  print 'Device: ', device.ip\_address, ' Interface: ', interface  print ''  print\_log(device\_ip, log\_info\_list) |

Chapter 16 Debugging, Testing and Logging

# 16.1 Introduction

When you have written your Python application, you will run it in order to determine if it is working as intended. Determining whether it is running correctly may involve merely looking at the output to see if the results are what was expected. Sometimes more thorough testing is required and this section will look into some mechanisms for taking testing to a deeper level. There are three important aspects of testing which will be considered:

* **Debugging:** Debugging an application means stopping execution in the middle of the application, and looking at the internal variables to see if they are as they should be. It can mean going through the code step-by-step in order to determine if it is operating as intended. It can mean tracing the thread of execution down into function calls in order to determine where and when a failure has occurred.
* **Unit testing:** Running your application and examining results is one way to determine a superficial level of quality. But a better way to test the completeness and quality of your application is via unit testing. Unit testing frameworks allow you to test the inputs and outputs of every facet of your code. Unit testing also provides a framework for creating automated tests through which you can insure quality across versions by performing regressions to make sure nothing that previously worked has been broken by the new code changes.
* **Logging:** When your application working in a live environment, you will not have the luxury of printing diagnostic messages to the console, because you typically will not be sitting at the console to observe them. Logging provides a mechanism for recording notable events (informational messages), potential issues (warning messages), problems (error messages), and failures (critical messages). These messages go into log files, which can later be examined to determine any issues that may be arising in your application's execution.

The chronology of these testing mechanisms generally follows the order in which they are presented here:

* **Development Phase:**Debugging is done during development, when you can open up your code to determine the internal defects that cause your program to misbehave.
* **Testing Phase:**Unit testing is built as a framework around your application, in part to insure that operational code does not get broken on subsequent builds and revisions of your application. Note that in software development, the idea of 'Test Driven Development' (TDD) has gained some popularity, wherein unit tests are written first, and the actual code to implement the feature or function is implemented in response to the unit tests that have been created. The unit tests that are described in this section can be used for TDD and for the more traditional develop-then-test model of creating software.
* **Production Phase:**Logging is put into your application for use primarily when it has been completed and is operating in 'production', meaning in a real-world environment. At this phase, it is difficult to go into debug mode, and writing and running tests is often not possible. Logging is the best way to understand the problem, in order to address it in some manner.

# 16.2 Debugging

Debugging is done most frequently during the development of your application. Two of the most popular methods of debugging are:

* **Print statements:** When writing your application, often the easiest way to test is by inserting print statements liberally within the code.
  + Traditional Method
  + Simple
  + def read\_devices(filen)
  + print 'reading: ', file
  + # code to read device
  + # information into list

print 'read input: ', list

* **Debugger:** Python, like other programming languages, provides a debugger, which allows you to examine the execution of your code without making any changes, such as adding print statements.
  + **pdb main.py**
  + No changes to your code
  + Line-by-line execution of your code and examination of your data
  + **Commands:** **next**, **step**, **return**, **list**, **set breakpoint**

## Printing

When you create your application you may not have any temporary print statements however, during debugging of potential issues or just for error checking, you may find them helpful. Here are some guidelines for where and how you may use temporary print statements to help with debugging your application:

* Print important events: read input file, created device objects, connected to a device, got interface data from a device, set interface description for a device, and so on.
* Print important data associated with events: contents of device file just read, attributes of device object just created, connection credentials, and so on.
* Print result codes from calls to external functions: from connecting to a device, making a database request, and so on.
* Print entry and exit from your own functions, in addition to printing the values of all the parameters.

Printing may only take you so far, however, and further inspection of the behavior of your application may require the use of a debugger.

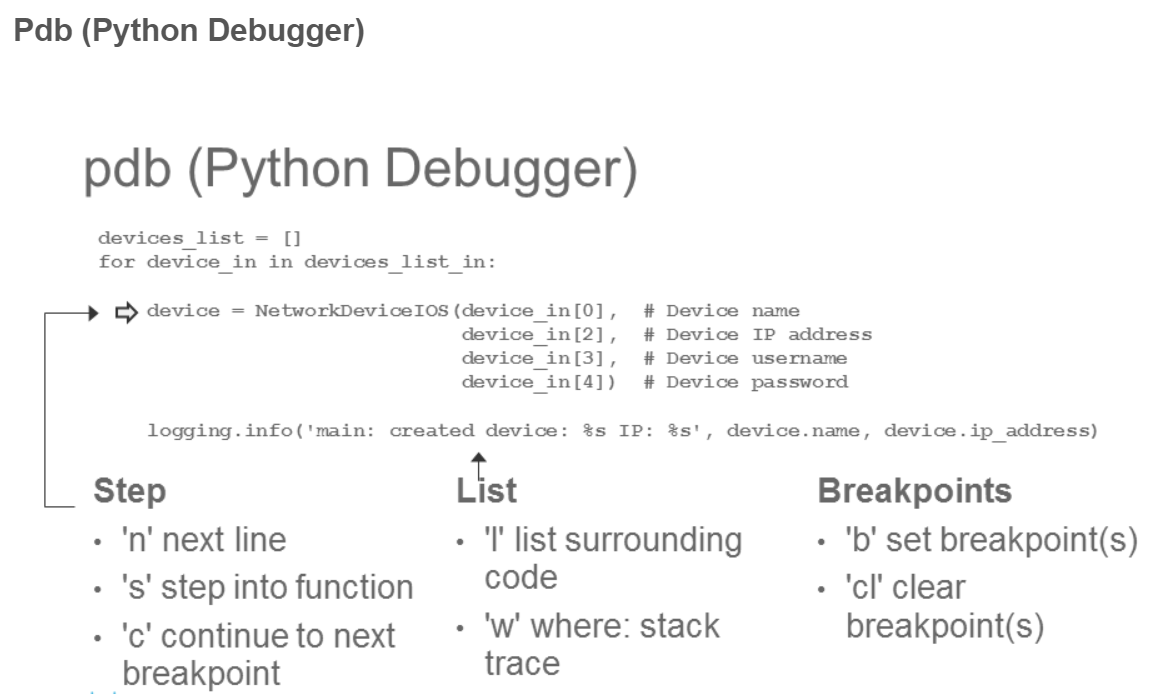
## Debugger

The process of using a debugger involves starting your application in a special debug environment. Started in this way, Python allows you to stop execution of your program in order to examine data, and to step through your program in order to determine the order and execution path your code is taking. Are your variables set to the values you think they should be? Is your for loop executing as you were expecting? A debugger will help you answer these questions.

Debugging commands fall into the following general categories:

* **Step-wise execution**: For the execution of lines one at a time, from your current line.
* **Code listing**: For looking at the lines of code surrounding your current line, and looking at the function-call history (also known as a 'stack trace').
* **Breakpoints**: For 'breaking' (stopping execution) at specific lines in your code.
* **Variables**: Looking at the values of variables, while stopped at your breakpoint.

## Pdb (Python Debugger)



The Python debugger **pdb** allows you to perform fine-grained debugging of your application.

To run the debugger, you simply invoke **pdb**rather than **python**:

$ pdb main.py

> /var/local/PyNE/labs/mod11/S11-2-logging/main.py(1)<module>()

-> import logging

It is also possible to run the debugger by specifying **pdb** as the module, and your application name as the script to use as input to the module:

$ python -m pdb main.py

> /var/local/PyNE/labs/mod11/S11-2-logging/main.py(1)<module>()

-> import logging

To step through code after starting the debugger, or after stopping at (also known as 'hitting') a breakpoint, you can use the following commands:

* **n**: Next – Execute the next line
* **s**: Step – Execute the next line, stepping into the function or method if applicable
* **c**: Continue – Continue on to the next breakpoint
* **r**: Return – Continue until the current function is about to return to the calling code.

You will likely not want to step through every line of your application, when attempting to debug a specific problem. In order to allow the execution of your application to continue up until a specific line of code, you will want to set a breakpoint. A breakpoint causes the debugger to halt execution of your application at a specific line, waiting for you to enter further instructions, such as to display variables, or to step through the code 1 line at a time.

Breakpoint commands are as follows:

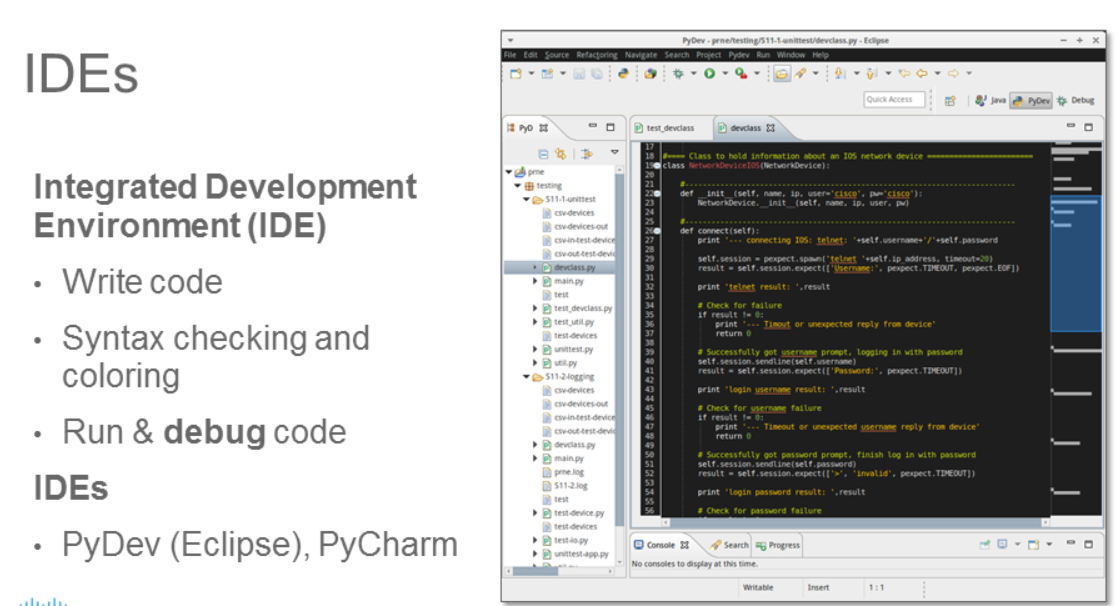
* **b**: Break– Set a breakpoint at the specific file and line number. It is also possible to set a conditional breakpoint, which breaks only if the provided condition is met. Specifying only the **b** with no arguments will list all breakpoints.
* **cl**: Clear – Clear specific breakpoints.

### Variables

When your application stops at a breakpoint, you will want to display information about the current state of your application: the line where you stopped, the calling modules, and the variables that are relevant now for your code. The following commands are available for this purpose:

* **l**: List – List the lines of code surrounding the location where the application is currently stopped. Notice that the current line at which your application is stopped has an arrow ('->') pointing to it.
* **w**: Where – List the stack trace of calling modules and functions that were called to get to a specific line of code.
* **p**: Print **–** Print the value of the variable or expression. Note that since the Python debugger is capable of interpreting code, you can list the variable name, and it will be printed as if you were running inside the Python interpreter.

## Integrated Development Environments (IDEs)



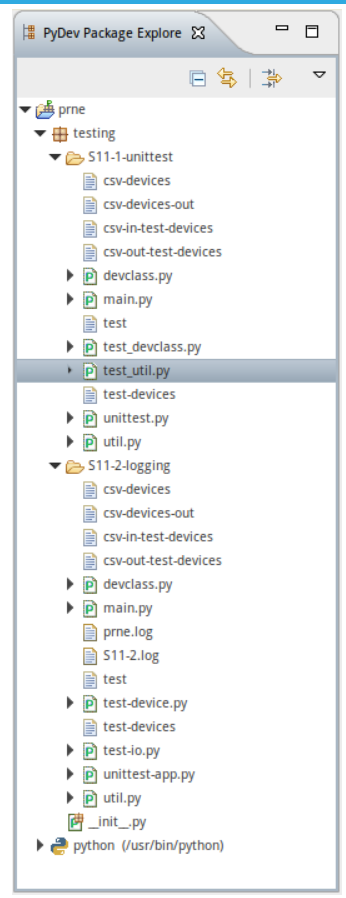
[IDE](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=IDE&type=terms)s provide a rich user interface and a significant functionality improvement over using the command line and general-purpose editors. IDEs integrate the management of projects, editing of software, and the running and debugging of applications, all in a powerful user interface.

The following features of IDEs make the development of applications much easier:

### Organizing

An IDE organizes your files and folders, your modules and packages, and presents them in an intuitive explorer-type format, for easy reference and access.

The example shows the Eclipse IDE with the 'PyDev' plugin, which provides Python-specific functionality. The figure is of the PyDev Package Explorer, showing the **prne** project, a package called 'testing', with Python modules and various other files.



### Coding

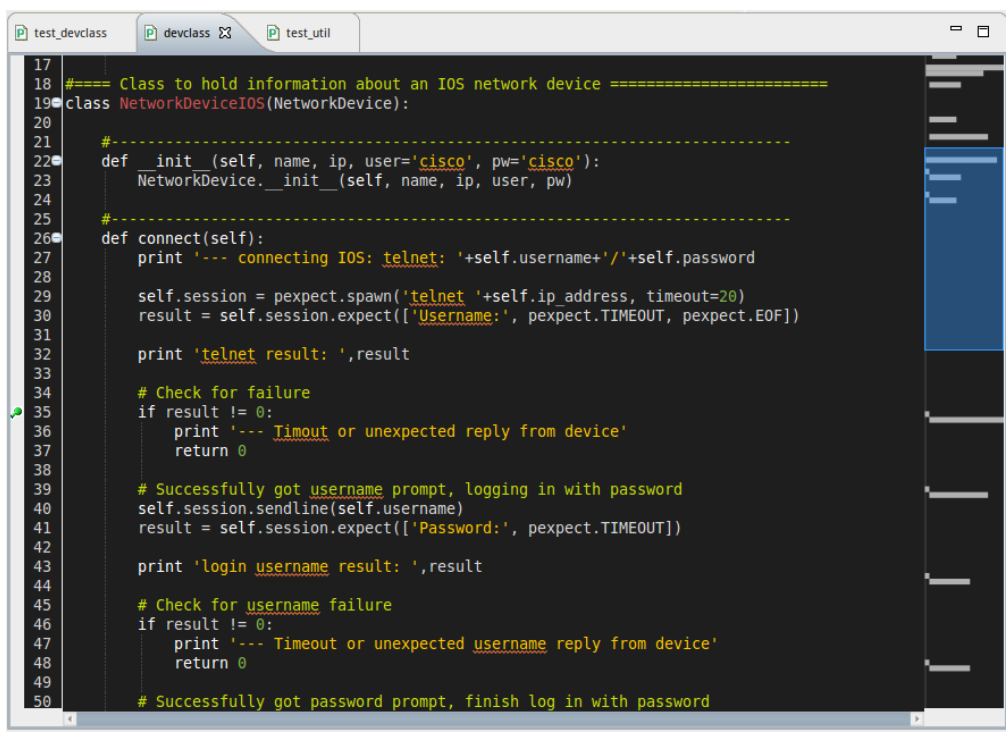
An IDE provides editors that are language-aware, in that they are able to analyze the code you write, highlighting and coloring keywords, strings, comments, and so on; maintaining consistent and automatic indentation as you type, and checking for syntax errors.

The following example shows the PyDev Python editor within Eclipse. You can see the different coloring highlighting comments, strings, indentation lines, and numeric values. Line numbers, which can be handy during debugging, are shown on the left.

### Debugging

In the figure above, the very left-most column shows breakpoints – the green dot identifies the line as having a breakpoint, and the check mark beside it indicates that the breakpoint is set correctly and is active.

Overall, IDEs are very useful tools for writing any type of software, Python included, and if your project is going to be of a moderate size or bigger, it would be wise to consider using such a tool.



# 16.3 lab Debugging a Network Application

In this lab, you will debug a Python application using **pdb**. The lab starts with a pre-written Python application that reads device information from a [CSV](https://ondemandelearning.cisco.com/cisco-cte/prne/search?query=CSV&type=terms) file, and then connects to each device in the network using Pexpect.

The provided application has three modules: **main.py**, **util.py**, and **devclass.py**. The application reads device information from a CSV file, creates device objects for every device, and then connects to every device.

Lab is currently initializing: 26% complete

## Learning Lab Procedure

**Step 1**

Run the application **main.py** using the python debugger. Open a terminal window and navigate to the Desktop\PRNE\section16\debugging folder. Run **main.py** using the python debugger.

### **Answer**

|  |
| --- |
| $ cd Desktop/PRNE/section16/debugging  $ pdb main.py |

**Step 2**

List the code surrounding your current location using the **list** command.

### **Answer**

You should see the following

|  |
| --- |
| (Pdb) **list**  1 -> from util import read\_devices\_info  2 from devclass import NetworkDeviceIOS  3  4 print '\n===== Reading CSV file, creating to devices'  5 devices\_filename = 'csv-devices'  6  7 devices\_list\_in = read\_devices\_info(devices\_filename) # read CSV info for all devices  8  9 # Iterate through all devices from the file, creating device objects for each  10 devices\_list = []  11 for device\_in in devices\_list\_in:  (Pdb) |

**Step 3**

Step to the next line using the **next** command. List the surrounding code.

### **Answer**

After issuing the next command, notice that the current line marker ('->') has changed to the second line of the application.

|  |
| --- |
| (Pdb) **next**  > /home/cisco/PRNE/section16/debugging/main.py(2)<module>()  -> from devclass import NetworkDeviceIOS  (Pdb) **list**  1 from util import read\_devices\_info  2 -> from devclass import NetworkDeviceIOS  3  4 print '\n===== Reading CSV file, creating to devices'  5 devices\_filename = 'csv-devices'  6  7 devices\_list\_in = read\_devices\_info(devices\_filename) # read CSV info for all devices  8  9 # Iterate through all devices from the file, creating device objects for each  10 devices\_list = []  11 for device\_in in devices\_list\_in:  (Pdb) |

**Step 4**

Set a breakpoint at the line calling the function **read\_devices\_info()**.

### **Answer**

The function **read\_devices\_info()** is being called on line 7. Use the command **break 7** to set the breakpoint.

|  |
| --- |
| (Pdb) **break 7**  Breakpoint 1 at /home/cisco/Desktop/PRNE/section16/debugging/main.py:7  (Pdb) |

**Step 5**

Resume execution until the breakpoint. List the code surrounding your current line to verify the breakpoint executed correctly.

### **Answer**

Use the **continue** command to resume execution.

|  |
| --- |
| (Pdb) **continue**  =====Reading CSV file, creating to devices  > /home/cico/Desktop/PRNE/section16/debugging/main.py(7)<module>()  -> devices\_list\_in = read\_devices\_info(devices\_filename) # read CSV info for all devices  (Pdb) **list**  2 from devclass import NetworkDeviceIOS  3  4 print '\n===== Reading CSV file, creating to devices'  5 devices\_filename = 'csv-devices'  6  7 B-> devices\_list\_in = read\_devices\_info(devices\_filename) # read CSV info for all devices  8  9 # Iterate through all devices from the file, creating device objects for each  10 devices\_list = []  11 for device\_in in devices\_list\_in:  12  (Pdb) |

**Step 6**

Step into the **read\_devices\_info()** function. List the code surrounding the current line. You are now inside the **util.py** module.

### **Answer**

Use the **step** command to step into the read\_devices\_info() function.

|  |
| --- |
| (Pdb) **step**  --Call--  >/home/cisco/Desktop/PRNE/section16/debugging/util.py(7)read\_devices\_info()  -> def read\_devices\_info(devices\_file):  (Pdb) **list**  2 from pprint import pprint  3  4 from devclass import NetworkDeviceIOS  5  6 #================================================  7 -> def read\_devices\_info(devices\_file):  8  9 devices\_list = []  10  11 file = open(devices\_file,'r') # Open the CSV file  12 csv\_devices = csv.reader(file) # Create the CSV reader for file  (Pdb) |

**Step 7**

Step through the function until you reach line 12 in the **util.py** module. List your code to verify that you are on the right line of code.

### **Answer**

You can use the **next** command or the **step** command to step through the function, executing one line of code at a time. You can also press enter.

|  |
| --- |
| (Pdb) **next**  >/home/cisco/Desktop/PRNE/section16/debugging/util.py(9)read\_devices\_info()  -> devices\_list = []  (Pdb) **next**  >/home/cisco/Desktop/PRNE/section16/debugging/util.py(11)read\_devices\_info()  -> file = open(devices\_file,'r') # Open the CSV file  (Pdb) **next**  >/home/cisco/Desktop/PRNE/section16/debugging/util.py(12)read\_devices\_info()  -> csv\_devices = csv.reader(file) # Create the CSV reader for the file  (Pdb) **list**  7 def read\_devices\_info(devices\_file):  8  9 devices\_list = []  10  11 file = open(devices\_file,'r') # Open the CSV file  12 -> csv\_devices = csv.reader(file) # Create the CSV reader for file  13  14 # Use list comprehension to put CSV data into list of lists  15 return [dev\_info for dev\_info in csv devices]  16  17 #================================================  (Pdb) |

**Step 8**

Use the **return** command to the main program (main.py). Note that the debugger takes you to the return statement in your current function. Use the **return** command again to arrive back at the calling code, then list the surrounding code.

### **Answer**

|  |
| --- |
| (Pdb) **return**  --Return--  > /home/cisco/Desktop/PRNE/section16/debugging/util.py(15)read\_devices\_info()  ->[['ios-01', 'ios', '10.30.30.1', 'cisco', 'cisco'], ['ios-02', 'ios', '10.30.30.2', 'cisco', 'cisco',]' ['ios-03', 'ios', '10.30.30.3', 'cisco', 'cisco]]  -> return [dev\_info for dev\_info in csv\_devices]  (Pdb) **return**  > /home/cisco/Desktop/PRNE/section16/debugging/main.py(10)><module>()  -> devices\_list = []  (Pdb) **list**  5 devices\_filename = 'csv-devices'  6  7 B devices\_list\_in = read\_devices\_info(devices\_filename) # read CSV info for all devices  8  9 # Iterate through all devices from the file, creating device objects for each  10 -> devices\_list = []  11 for device\_in in devices\_list\_in:  12  13 device = NetworkDeviceIOS(device\_in[0], # Device name  14 device\_in[2], # Device IP address  15 device\_in[3], # Device username  (Pdb) |

**Step 9**

Display the contents of **devices\_list\_in** to make sure that the code has correctly read the device information from the CSV file.

### **Answer**

Use the print command to display the contents of the variable.

|  |
| --- |
| (Pdb) **print devices\_list\_in**  [['ios-01', 'ios', '10.30.30.1', 'cisco', 'cisco'], ['ios-02', 'ios', '10.30.30.2', 'cisco', 'cisco',]' ['ios-03', 'ios', '10.30.30.3', 'cisco', 'cisco]]  (Pdb) |

**Step 10**

Use the **continue** command to execute the rest of the program.

### **Answer**

|  |
| --- |
| (Pdb) **continue**  ----- created device: name: ios-01 IP: 10.30.30.1  ----- created device: name: ios-02 IP: 10.30.30.2  ----- created device: name: ios-03 IP: 10.30.30.3  ===== Connecting to each device  ----- connected to device: ios-01 IP: 10.30.30.1  ----- connected to device: ios-02 IP: 10.30.30.2  ----- connected to device: ios-03 IP: 10.30.30.3  The program finished and will be restarted  > /home/cisco/Desktop/PRNE/section16/debugging/main.py(1)<module>()  -> from util import read\_devices\_info  (Pdb) |

**Step 11**

Remove the breakpoint and exit the python debugger.

### **Answer**

|  |
| --- |
| (Pdb) **clear**  Clear all breaks? **y**  (Pdb) **quit**  $ |