LABRST-2678

Programming for Network Engineers

Ambili Sasidharan – CX Technical Leader
Senthil Palanisamy – CX Architect
Jagadesh Nagaraj – CX Consulting Engineer

Table of Contents

Learning Objectives

Upon completion of this lab, you will be able to:

- Understand various use cases and examples of network programmability and automation.
- Understand the basic concepts on how to build python scripts and automate network tasks.
- Interact with Cisco devices using python programming language.
- Use python SSH libraries to communicate with cisco devices.

Prerequisites

- Basic Python programming, Linux and REST API knowledge.
- It is strongly recommended to read through the lab guide and complete all lab tasks, which will give you a good foundation for understanding automation concepts and interacting with cisco devices using Python.

Disclaimer

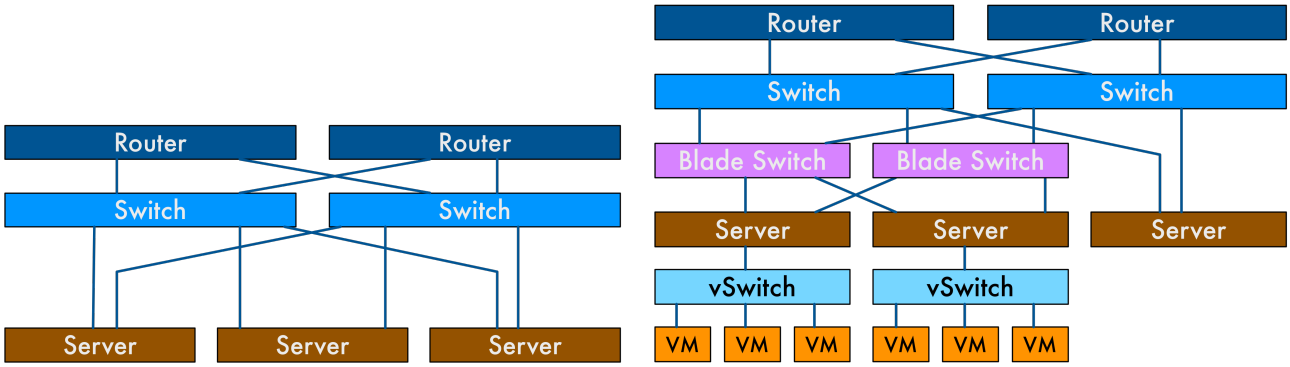
This training document is to get you familiarize with network programming and automation concepts. Although the lab design, configuration and python demo scripts could be used as a reference, it's not a real design, thus not all recommended features are used, or enabled optimally to use it in production networks. For the design related questions please contact your representative at Cisco, or a Cisco partner. Else feel free to reach out to us. We will be glad to answer.

Pre-Configuration

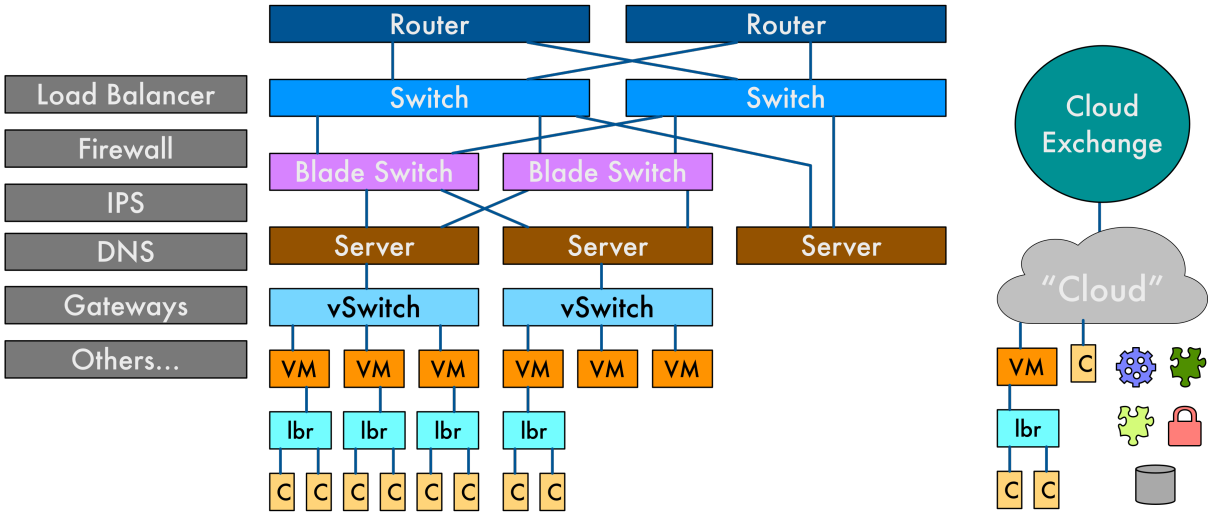
Due to limited duration of WISP lab, we have the devices preconfigured to save your time.

Introduction

In the SDN Era, the roles and responsibilities of the Network Engineers are drastically changing. Reducing Opex and Faster Time to market are the key business drivers for most of the IT organizations today. As the networks grow in size, it has increased the network management and operations complexity to a very high level. The state of the network has changed a lot over the recent years from a simple three tier architecture to multi-level modern network architecture. Especially the server virtualization and container technologies has revolutionized the landscape of IT.



Simple Network



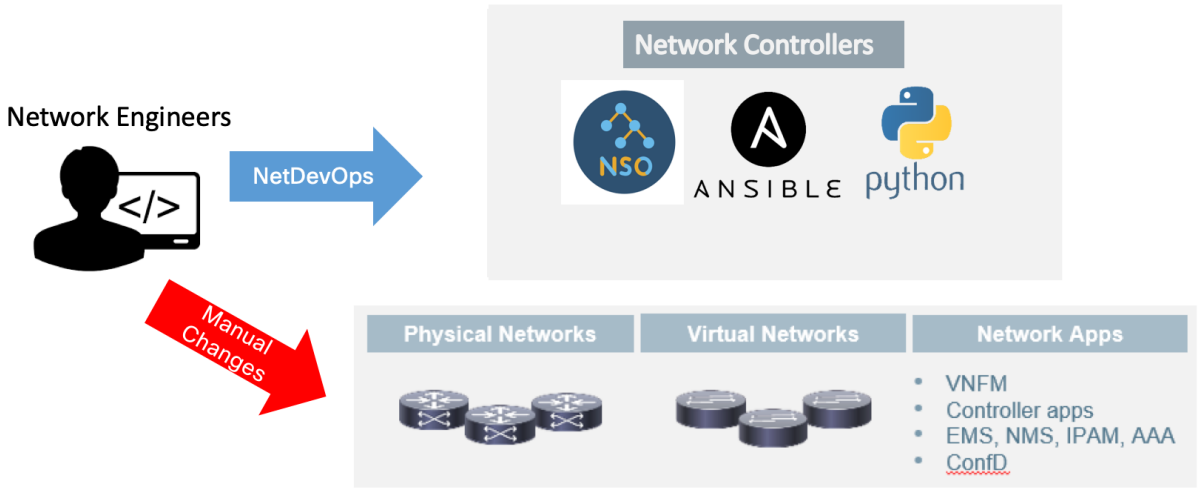
Modern Network

Network Programmability is the new way of communicating with the network devices. In today's world network administrators want programming capabilities in their networking devices for several reasons. The main reason is to automate provisioning, push configuration changes and troubleshoot complex network issues. Additionally, leveraging automation leads to a more predictable and uniform network as a whole and streamlines the process for all users supporting a given network environment.

Manual vs Automation

CLI is still a critical tool in the network engineers tool kit, because it is readily available, easy to use and grants valuable insights about the state of network. Unfortunately, CLI is not without its flaws, especially when it comes to network programmability and automation. CLI is inherently not scalable and it is designed to have 1:1 human interaction which makes it slow. Another issue is that the CLI outputs or data passed over CLI are unstructured raw data which are traditionally designed in a human readable fashion. This makes screen scraping increasingly complex when interacting with multiple different hardware platforms and software versions.

As today's networks are constantly growing in size every day, manually making changes or managing it will soon become a huge challenge. So the solution is to adopt NetDevops concepts for managing the network and move away from making manual changes to the network. By moving towards a devops model to manage the network, we can build networks that can scale better, reduce human errors, reduce configuration drift and faster troubleshooting. Consistency, Reliability and Scalability are the three main pillars that defines the modern network. The general rule here is Use Programmability and Automation whenever possible and fallback to using CLI for the tasks that cannot be automated.
.



The analogy here is that you don't need to know all the parts that goes inside a car to drive a car. This lab is designed in the same way, as a Network Engineer who is getting started with programming you don't need to spend countless hours of time in educating yourself with learning all the basics of programming. Instead create a lab infrastructure, leverage the work that is already done by the networking community, play and learn along the way. Start with collecting show command outputs, inventory details and making small changes that is not going to break your network. For Ex: changing hostname, add syslog or ntp servers, add dummy loopback and acl's.

Why Python

Python is a very powerful and easy to learn programming language. Python Programming skill is mandatory for today's Network engineers, one needs to know and comfortable with how to read, write, edit, modify and expand python scripts. It is specifically around the operations of a network that learning to read and write some code starts to make sense.

The way we manage the network haven't changed much for many years. As network engineers we have done the same repeated tasks thousands of time over and over again. Using python programming we can automate most of the repeated network admin tasks any number of times without any typo or human errors. Another benefit is you can save lot of time which is spent on doing boring repeated/manual network admin tasks by automating it.

The main benefit of using python for network management is that, we don't need to reinvent the wheel. There are lot of open-source and community driven python libraries that we can leverage to quickly get started and bring our self upto speed in a very short time. There are now network device APIs, community-supported Python libraries and various open source tools that will help a lot to jump start your network automation skillset.

There are two types of tasks that you can try automating:

1) Frequently done small tasks
2) One-time bulk changes

Again, please remember "With great power comes great responsibility", Error prone scripts can break the network very fast and can cause major business impact. So, the advice here is to build and test the python scripts in a safe lab environment first, not once but multiple times, before trying it out in production.

For this lab we will be using two very popular python library named Netmiko.

Netmiko is an open-source python library based on Paramiko SSH library. It provides a uniform programming interface across a broad set of network devices. It also handles many of the low-level SSH details that can be time consuming and problematic when you are a beginner and getting started with programming.

The purpose of this library is:

- Successfully establish an SSH connection to the device
- Simplify the execution of show commands and the retrieval of output data
- Simplify execution of configuration commands
- Support for multivendor networking platforms

NAPALM (Network Automation and Programmability Abstraction Layer with Multivendor support) is a Python library that implements a set of functions to interact with different network device Operating Systems using a unified API. NAPALM supports several methods to connect to the devices, to manipulate configurations or to retrieve data across several network operating systems, including Cisco IOS, IOS-XE, IOS-XR, NX-OS and other vendor platforms.

Primary functions of NAPALM:

- Gather facts from networking devices
- Configuration management
- Configuration rollback

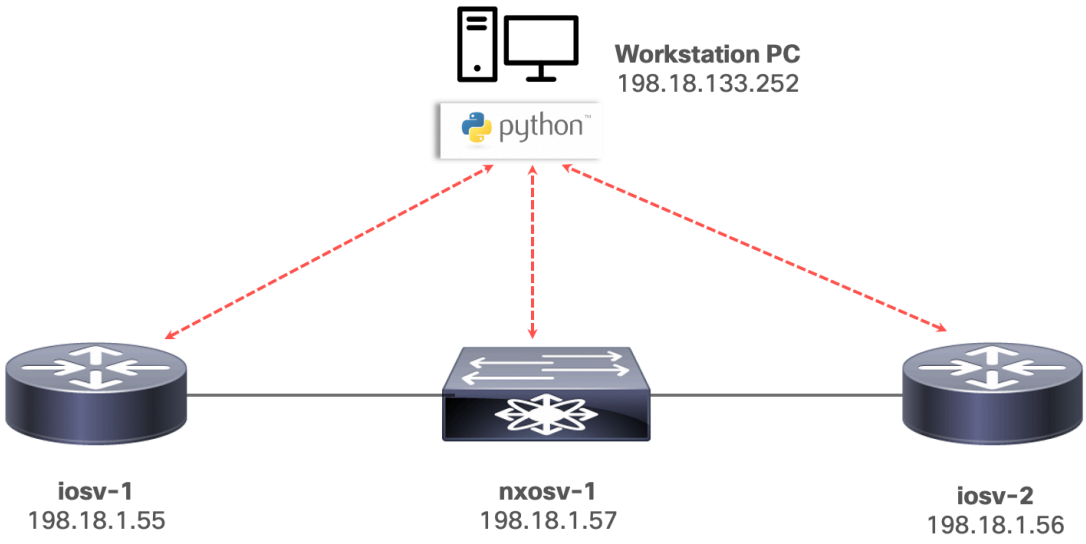
There are several other features and functionalities available in Netmiko and NAPALM, we are not covering all of it. Similarly, there are lot of vendor and community developed python libraries and SDK's that you might come across which does similar functions too. The goal here is to educate Network engineers about how easy it is to get started with network programmability and automation without going too much deeper into python programming concepts.

For beginners learning about programming and automation can be very intimidating. Don't worry there are many sessions in Ciscolive that you can attend and get yourself educated. Please check the Devnet zone, breakout session catalog and reference section to learn more about Python programming, Network Programmability and Automation.

Some of the recommended Devnet sessions that you can attend this week are:
Coding 1001 - Intro to APIs and REST - DEVNET-1897
Coding 1002 - Getting Started with Python - DEVNET-1893

Note: Search for "devnet coding" or "devnet python" on youtube. You can get access to the previous recording of the coding sessions.

Lab Topology



Note: The workstation provides all the required access to the lab devices

Connection Details:

Hostname	IP	Username	Password	Connection	Port
iosv-1	198.18.1.55	cisco	cisco	SSH	22
iosv-2	198.18.1.56	cisco	cisco	SSH	22
nxosv-1	198.18.1.57	cisco	cisco	SSH	22
Workstation	198.18.133.252	Administrator	C1sco12345	RDP	3389

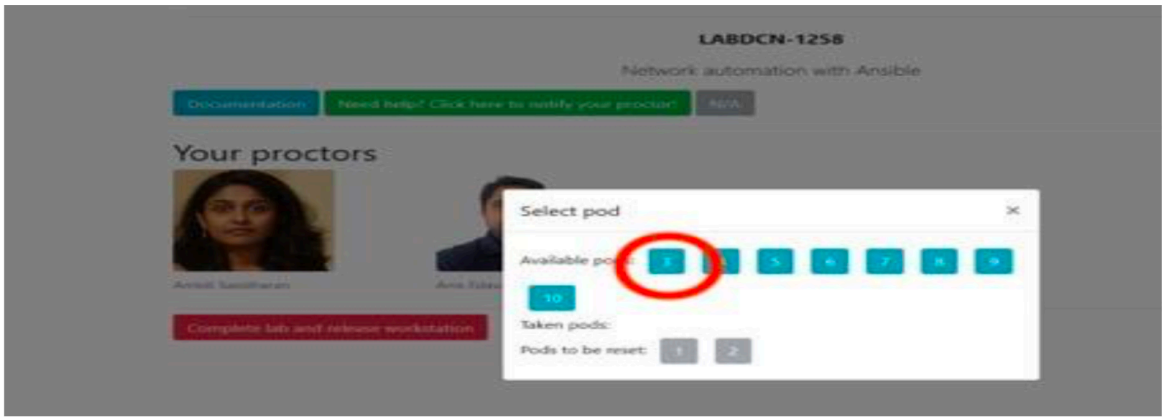
Code repository:

https://github.com/jagadnag/cl_python_2020

Connecting to the WIL Assistant and accessing the Lab

Step 1: From the WIL assistance home page @ www.wilassiatance.com , click on START and search for the lab 2658 and click on START. You will see the page as below. Click on the buton N/A to select the Pod.

Step 2: Please select the Pod that is assigned to you on the hand out given to you or the lab proctor.



Step 3: Open the lab guide and follow the lab guide to connect to the RDP session and proceed to the lab exercises.

Connect to dCloud lab network

This lab leverages dCloud, which uses virtual servers and switches that runs on a simulator. These devices have most of the functions of actual hardware based devices, but there is no data plane traffic.

You need to VPN into the dCloud environment in order to access devices and complete this lab.

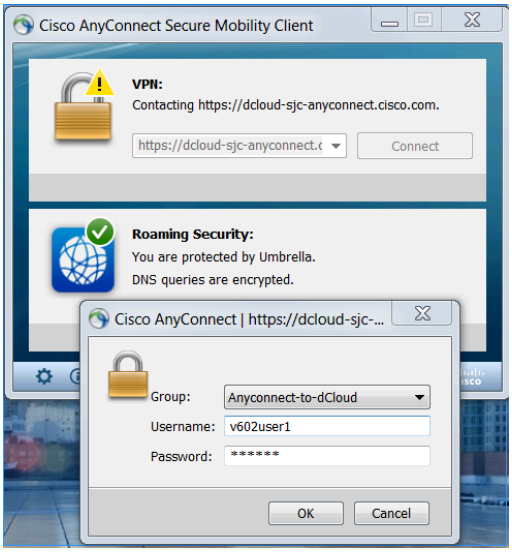
Click on the Cisco AnyConnect client and check to see if the client is connected to another VPN session. If yes, go ahead and disconnect.

Now type or copy paste the VPN server details:

dcloud-rtp-anyconnect.cisco.com

For the vpn credentials, use the username and password that is attached to the lab card. Input the username and password. Click OK.

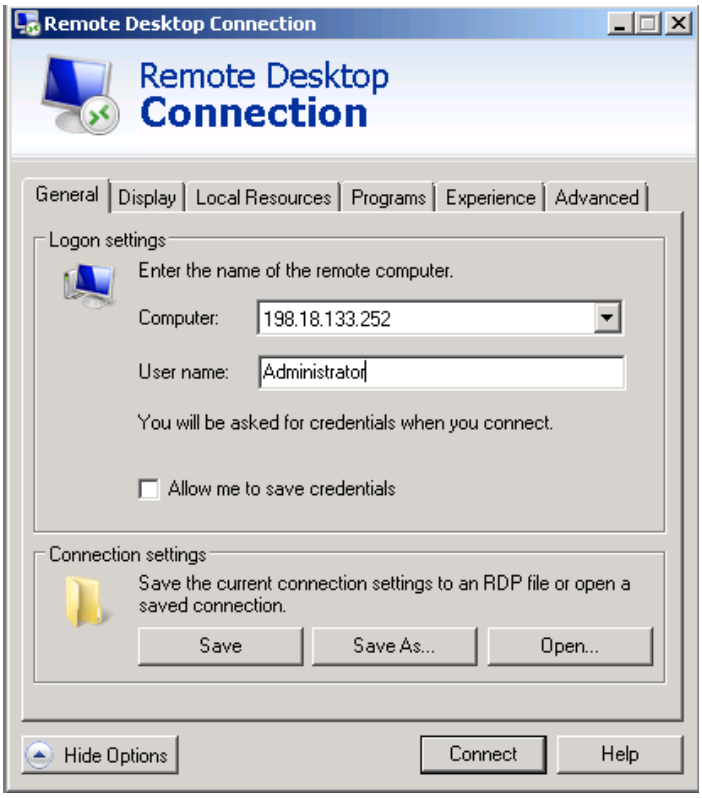
If you need any additional assistance, please use the WISP lab assistant to get help from lab proctors.

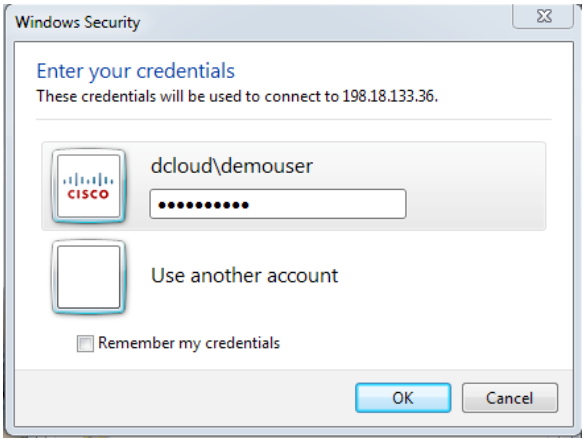


Once VPN connection is successful, you should be able to access the lab environment.

Now open a Remote Desktop Connection (RDP) to the lab workstation using the RDP client.

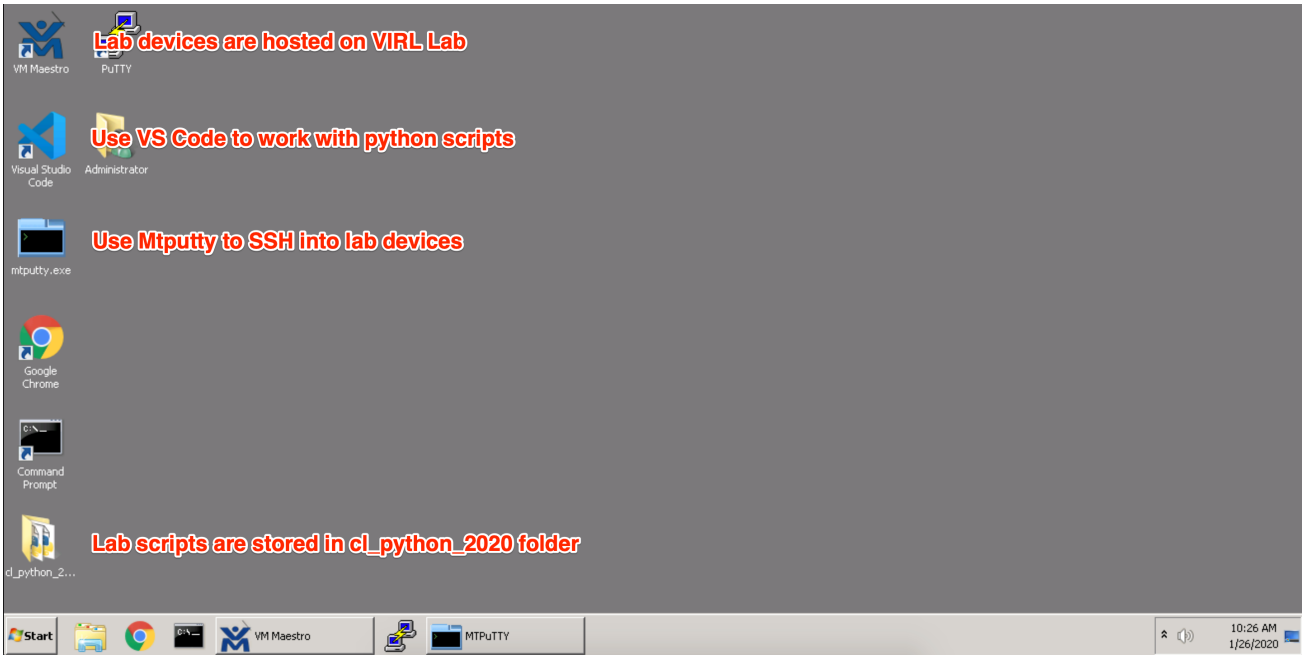
Workstation
IP Address:
198.18.133.252
Credentials:
username: Administrator
password: C1sco12345



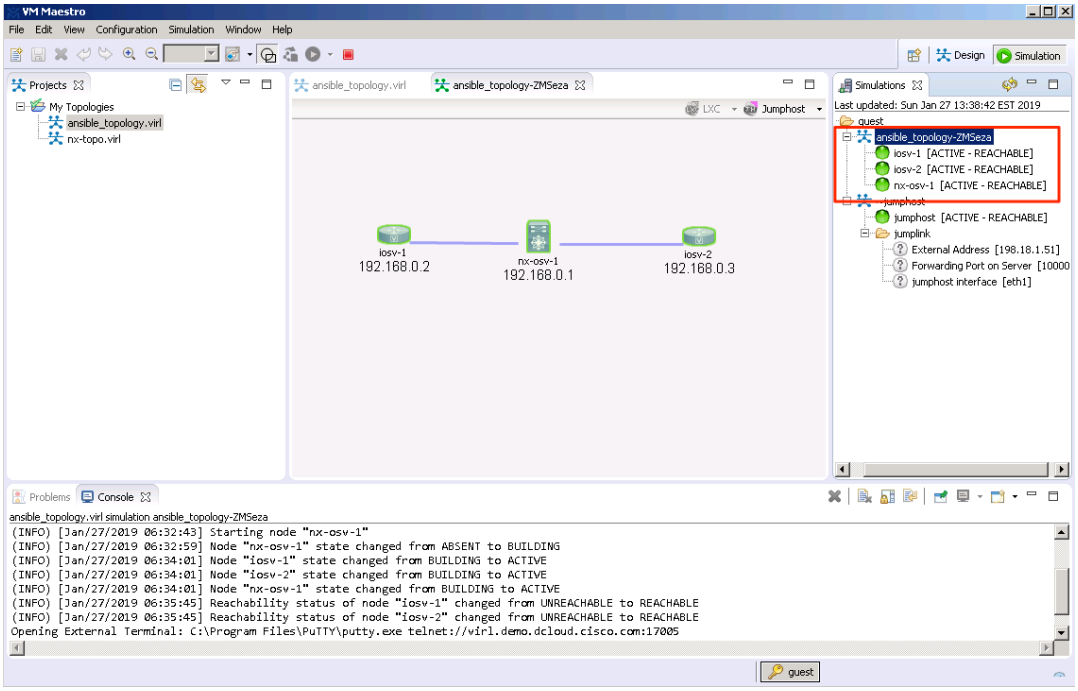


Enter the password for RDP connection – C1sco12345

After establishing the RDP connection, you will see a similar desktop screen setup as shown below. Use the tools as needed to complete the lab tasks.



Open VM Mastero application and ensure all the lab devices are in green (active state). You can also use mtputty to ssh into the lab devices and check their status.



By default, beginner programmers will use python shell and IDLE for learning purposes. Once after getting some hands-on experience with IDLE, you will be using a Code editor to develop scripts and test python scripts. For this lab we will be using Microsoft Visual Studio (VS) Code software which is very user friendly and feature rich. It is available for macOS, Linux, and Windows operating systems. For more information, please refer to: https://code.visualstudio.com/

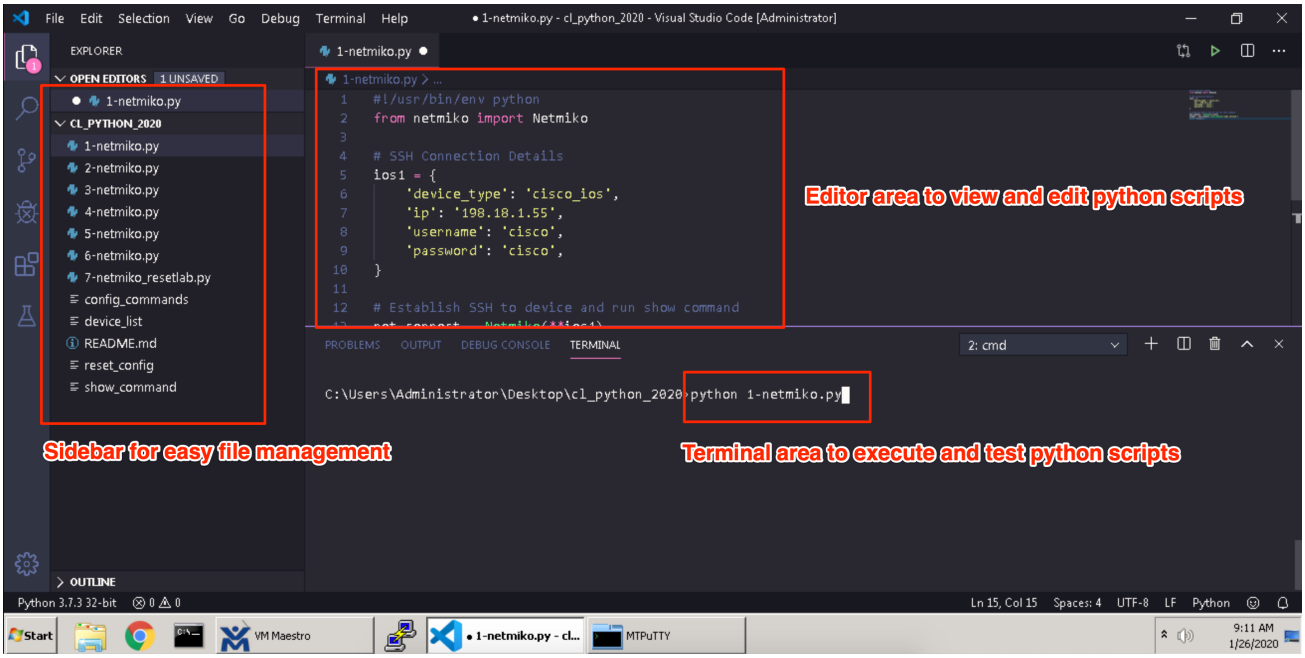
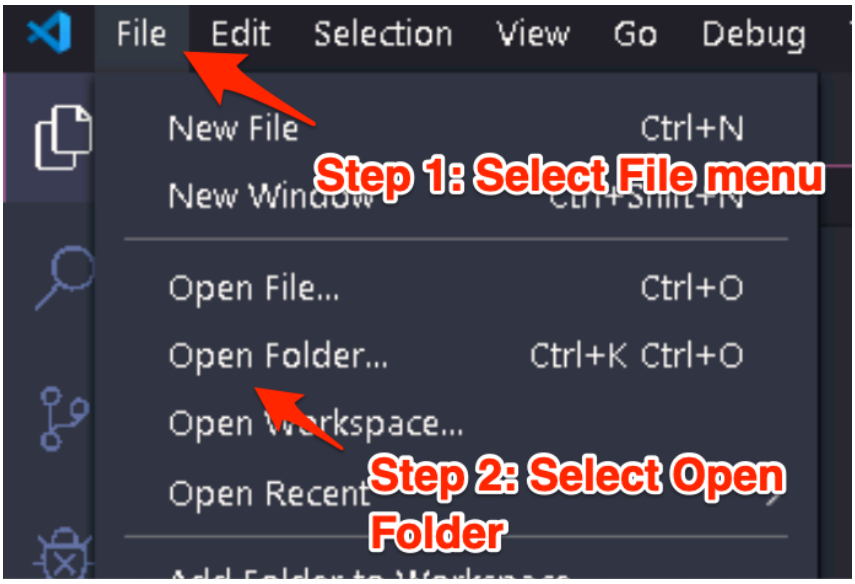
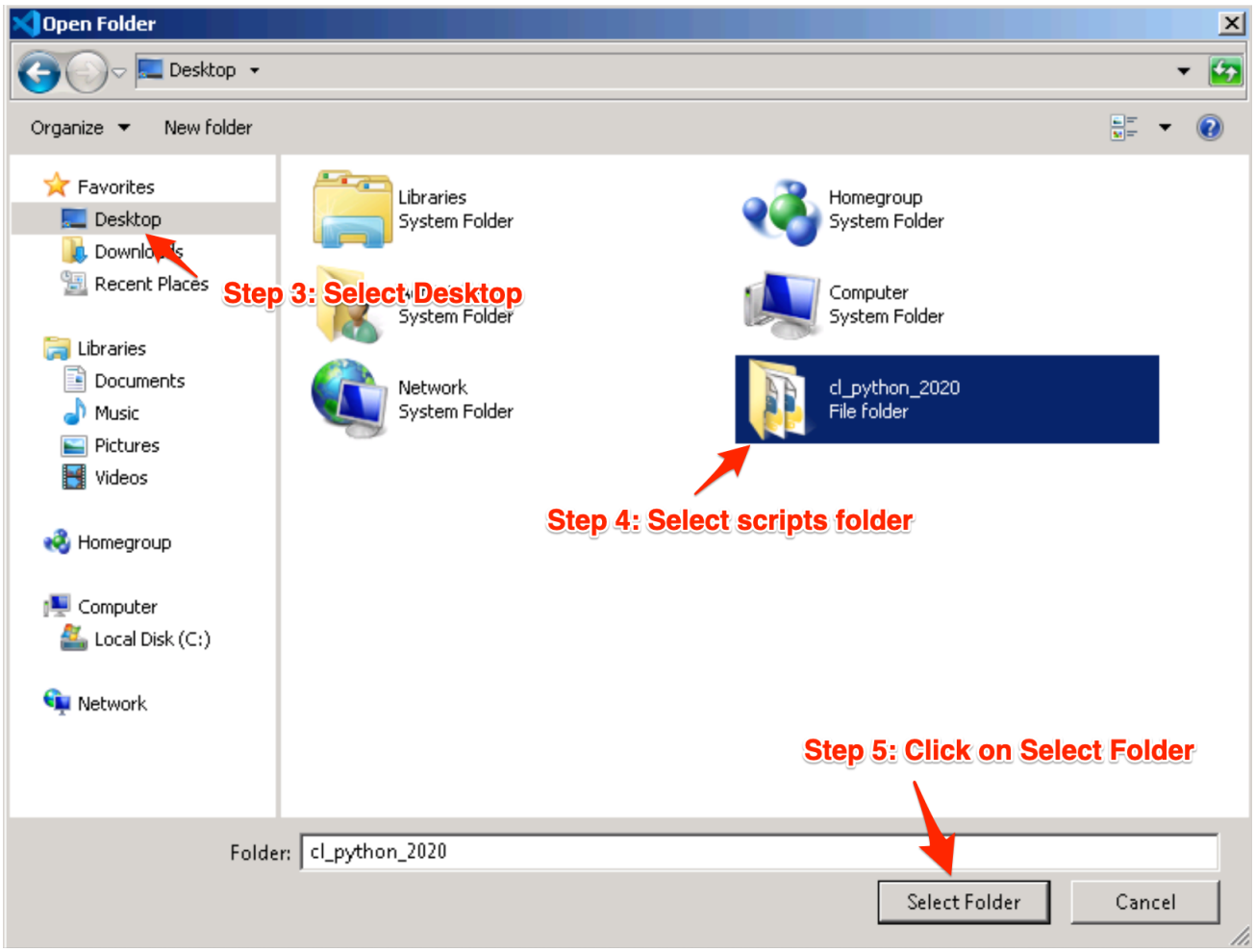


Figure: Visual Studio Code Editor Layout

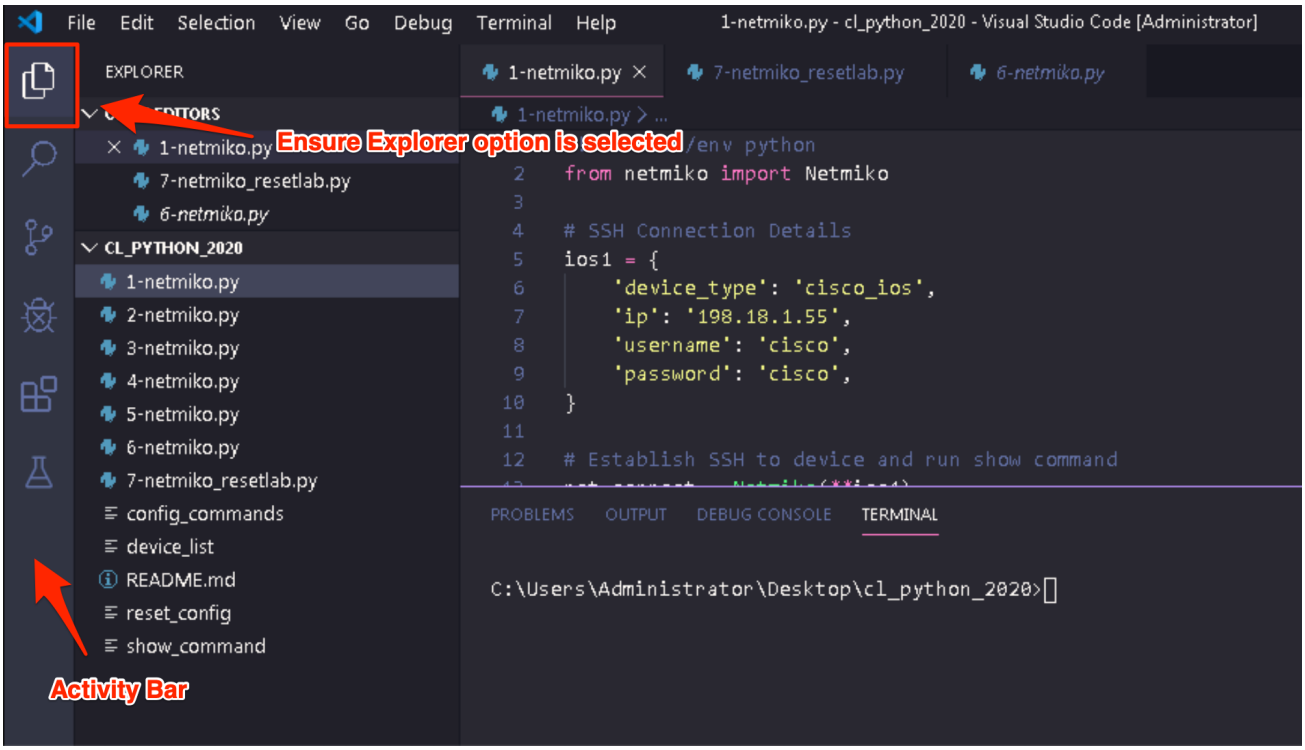
Here are some tips that will get you started with VS Code editor.

How to open the scripts folder:

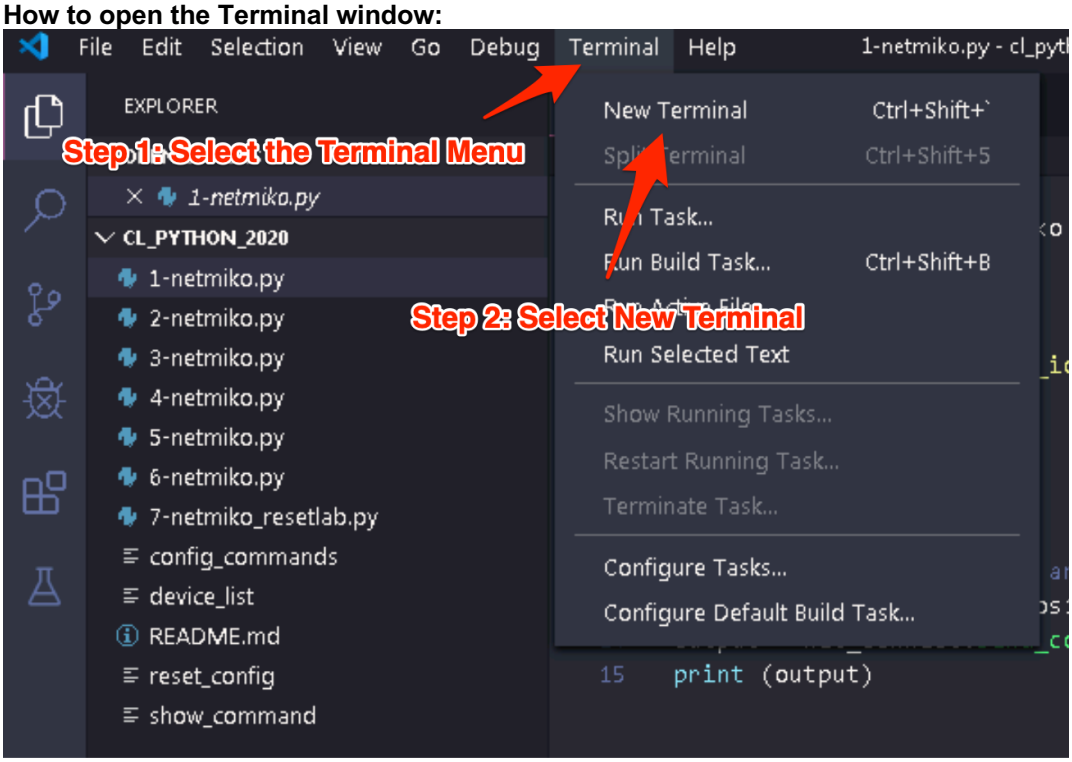




Ensure Explorer option is selected



How to open the Terminal window:



How to execute python scripts from VS Code:

To execute the scripts, go to the Terminal window, type “python” and name of the script and press enter. If python requests you to enter some inputs (for ex: username or password) type the requested information and press enter.

***Note**: Please check the folder path, if you are not in the right folder where the scripts are stored. Python will throw a “No such file or directory error”. So please verify the path first before executing the scripts.*

Lab Tasks

Building blocks of Netmiko python script

In this lab exercise you will learn how a basic netmiko python script is constructed. Think of it like a Lego block, first we will start with basic building blocks and quickly iterate through it to build complex programming logics.

Let's dive into the basic building blocks.

First we have to import the Netmiko connection libraries

```
from netmiko import Netmiko
```

The connection parameters are collected in a Python dictionary. The connection parameters provide Netmiko with everything it needs to create the SSH connection. In the below example we have shown how to connect to an IOS device, if it is a NXOS device, simply change the device type to 'device_type': 'cisco_nxos'.

```
ios1 = {
    'device_type': 'cisco_ios',
    'ip': '192.18.1.55',
    'username': 'cisco',
    'password': 'cisco',
}
```

Netmiko is a function that calls the necessary connection parameters and device type (cisco_ios, cisco_xr, cisco_nxos, etc.) Once connection parameters are loaded, the script will launch a SSH connection to login into the device.

```
net_connect = Netmiko(**ios1)
```

.send_command() method is used to send show commands over the channel and receive the output back. Here, we are reading the output of 'show version' command and storing it in a variable named 'output'.

```
output = net_connect.send_command('show version')
```

Using the .send_config_set() method, we can program the network device to enter into configuration mode and make configuration changes. After executing the config_commands the script will exit the configuration mode.

```
output = net_connect.send_config_set(config_commands)
```

We can send either only one command or multiple lines of commands by converting it into a simple list. If we are sending a big configuration, it is recommended to use the .send_config_from_file() method.

```
output = net_connect.send_config_from_file('more_config')
```

All of the session output is stored in an output variable and then printed out in the screen for our reference.

For more information on the all the connection methods available with Netmiko, pls refer the documentation

[Netmiko Introduction]

[Netmiko Documentation]

Task 1: Executing a show command on a network device

The python script will

- login to the iosv-1 device via SSH
- run the show version command
- capture the output
- print the command output to the screen.

```
#!/usr/bin/env python
from netmiko import Netmiko

# SSH Connection Details
ios1 = {
    'device_type': 'cisco_ios',
    'ip': '198.18.1.55',
    'username': 'cisco',
    'password': 'cisco',
}

# Establish SSH to device and run show command
net_connect = Netmiko(**ios1)
output = net_connect.send_command('show version')
print (output)
```

Task 2: Configuring a network device

The python script will

- login to the iosv-1 device via SSH
- make a configuration change "logging host 10.1.1.1"
- capture the output
- print the command output to the screen.

Notice we are reusing the same code, that we used in previous task. Only change is, instead of net_connect.send_command() we are using net_connect.send_config_set(). Rest of the script remains the same. Moving forward we will use the same logic, iterate through the code we have already written and add additional logic on top of it to automate complex network tasks.

```
#!/usr/bin/env python
from netmiko import Netmiko

# SSH Connection Details
ios1 = {
    'device_type': 'cisco_ios',
    'ip': '198.18.1.55',
    'username': 'cisco',
    'password': 'cisco',
}

# Establish SSH to device and run config command
net_connect = Netmiko(**ios1)
output = net_connect.send_config_set('logging host 10.1.1.1')
print (output)
```

Task 3: Configuring multiple network devices

The python script will

- login to the iosv-1 and iosv-2 devices via SSH
- make a configuration change "logging host 10.1.1.2"
- capture the output
- print the command output to the screen.

To configure multiple devices, we have to create multiple SSH connection profiles and add it to a list. Then add a for loop to iterate through the connection profiles and make config changes to the IOS devices.

```
device_list = [ios1, ios2]
for device in device_list:
    ** Netmiko config code block **
```

```
#!/usr/bin/env python
from netmiko import Netmiko

# SSH Connection Details
ios1 = {
    'device_type': 'cisco_ios',
    'ip': '198.18.1.55',
    'username': 'cisco',
    'password': 'cisco',
}

ios2 = {
    'device_type': 'cisco_ios',
    'ip': '198.18.1.56',
    'username': 'cisco',
    'password': 'cisco',
}

devices = [ios1, ios2]

for device in devices:
    # Establish SSH to device and run config command
```

```
    print ('Connecting to device ' + device['ip'])
    net_connect = Netmiko(**device)
    output = net_connect.send_config_set('logging host 10.1.1.2')
    print (output)
```

Task 4: Pushing large configurations across multiple devices

Now we have a solid understanding of how to write basic scripts. Next step is to make our code modular. For that we should remove all of the hardcoded variables from the script. The variables that are used in the script has to be provided by the user who runs the script or from an external file.

The python script will

- Load the device ip details from a text file named 'device_list'
- Load the configuration commands from a text file named 'config_commands'
- Requests the login credentials from the user
- Uses getpass() module to encrypt the user provided password
- Login to each device in the 'device_list' and configure the commands given in 'config_commands' file.
- Print the output

First we use the input() and getpass() modules to collect the login credentials. Next, read the contents of the file using inbuilt python file module - with open(). After that we loop through the device_list and configure the devices.

```
username = input()
password = getpass()

with open('file_name') as f:
    device_list = f.read().splitlines()

for devices in device_list:
    ** Netmiko config code block **
```

```
#!/usr/bin/env python
from netmiko import Netmiko
from getpass import getpass

# SSH username and password provided by user
username = input('Enter your SSH username: ')
```

```
password = getpass('Enter your password: ')

# Sending device ip's stored in a file
with open('device_list') as f:
    device_list = f.read().splitlines()

# Iterate through device list and configure the devices
for device in device_list:
    print ('Connecting to device ' + device)
    ip_address_of_device = device

    # SSH Connection details
    ios_device = {
        'device_type': 'cisco_ios',
        'ip': ip_address_of_device,
        'username': username,
        'password': password
    }

    net_connect = Netmiko(**ios_device)
    output = net_connect.send_config_from_file('config_commands')
    print (output)
```

Task 5: Error handling and verification

In this task we will demonstrate how to enable error handling for our scripts. The idea behind error handling is to catch any exceptions that occurs during the execution of the script. Without error handling when an exception is detected the python script terminates and reports error.

Try and expect code blocks will help us to catch any errors during program execution. We have added different exceptions that can be triggered during the execution. For example: device timeout, reachability issues, wrong user credential errors, etc. If an exception is detected, the script will move on to the next device and complete the task.

```
username = input()
password = getpass()

with open('file_name') as f:
    device_list = f.read().splitlines()

for devices in device_list:
    try:
        ** Netmiko connection **
    except:
        ** Error condition **
        continue
    ** Netmiko config code block **
```

Test the exceptions, try providing wrong username and password to check whether the scripts catch and report the exceptions.

```
#!/usr/bin/env python
from getpass import getpass
from netmiko import Netmiko
from netmiko.ssh_exception import NetMikoTimeoutException
from paramiko.ssh_exception import SSHException
from netmiko.ssh_exception import AuthenticationException

# Collect login credentials
username = input('Enter your SSH username: ')
```

```
password = getpass('Enter your password: ')

# Sending device ip's stored in a file
with open('device_list') as f:
    device_list = f.read().splitlines()

# Iterate through device list and configure the devices
for devices in device_list:
    print ('Connecting to device ' + devices)
    ip_address_of_device = devices
    ios_device = {
        'device_type': 'cisco_ios',
        'ip': ip_address_of_device,
        'username': username,
        'password': password
    }
    # Error handling parameters
    try:
        net_connect = Netmiko(**ios_device)
    except (AuthenticationException):
        print ('Authentication failure: ' + ip_address_of_device)
        continue
    except (NetMikoTimeoutException):
        print ('Timeout to device: ' + ip_address_of_device)
        continue
    except (EOFError):
        print ("End of file while attempting device " +
ip_address_of_device)
        continue
    except (SSHException):
        print ('SSH Issue. Are you sure SSH is enabled? ' +
ip_address_of_device)
        continue
    except Exception as unknown_error:
        print ('Some other error: ' + str(unknown_error))
        continue
```

```
    # Configure the device and save config
    output = net_connect.send_config_from_file('config_commands')
    output += net_connect.send_command('wr mem')
    print (output)
```

Task 6: Show command verification

In this task we will reuse the previous script to collect multiple command outputs. This is one classic example of a task that you can automate when you are running change windows. You can use this script as pre-check and post-check command output validation.

```
#!/usr/bin/env python
from getpass import getpass
from netmiko import Netmiko
from netmiko.ssh_exception import NetMikoTimeoutException
from paramiko.ssh_exception import SSHException
from netmiko.ssh_exception import AuthenticationException

# Collect login credentials
username = input('Enter your SSH username: ')
password = getpass('Enter your password: ')

# Sending device ip's stored in a file
with open('device_list') as f:
    device_list = f.read().splitlines()

# Sending list of show commands stored in a file
with open('show_command') as f:
    show_commands = f.readlines()

# Iterate through device list and configure the devices
for devices in device_list:
    print ('Connecting to device ' + devices)
    ip_address_of_device = devices
    ios_device = {
```

```
        'device_type': 'cisco_ios',
        'ip': ip_address_of_device,
        'username': username,
        'password': password
    }
    # Error handling parameters
    try:
        net_connect = Netmiko(**ios_device)
    except (AuthenticationException):
        print ('Authentication failure: ' + ip_address_of_device)
        continue
    except (NetMikoTimeoutException):
        print ('Timeout to device: ' + ip_address_of_device)
        continue
    except (EOFError):
        print ("End of file while attempting device " +
ip_address_of_device)
        continue
    except (SSHException):
        print ('SSH Issue. Are you sure SSH is enabled? ' +
ip_address_of_device)
        continue
    except Exception as unknown_error:
        print ('Some other error: ' + str(unknown_error))
        continue

    # Iterate through command list and print the output
    net_connect = Netmiko(**ios_device)
    for command in show_commands:
        output = net_connect.send_command(command)
        print (command + output + '\n')
```

Building blocks of NAPALM python script

In this exercise you will learn how a basic NAPALM python script is constructed.

First we have to import the NAPALM connection drivers

```
from napalm import get_network_driver
```

The connection parameters are provided to the network driver, which includes end device OS type, ip address and login credentials.

```
driver = get_network_driver('ios')

device = driver('198.18.1.55', 'cisco', 'cisco')
```

NAPALM will open the SSH connection using the connection parameters

```
device.open()
```

Once connection is established, NAPALM functions are called to perform various tasks.

```
device.get_facts()
```

Close the SSH connection.

```
device.close()
```

Primary functions of NAPALM

- **load_merge_candidate:** Populate the candidate config, either from file or text.
- **load_replace_candidate:** Similar to load_merge_candidate, but instead of a merge, the existing configuration will be entirely replaced with the content of the file, or the configuration loaded as text.
- **compare_config:** Return the difference between the running configuration and the candidate.
- **discard_config:** Discards the changes loaded into the candidate configuration.
- **commit_config:** Commit the changes loaded using load_merge_candidate or load_replace_candidate.
- **rollback:** Revert the running configuration to the previous state.

Additional get functions

- **get_facts:** collect facts and operational data from end devices (vendor, model, uptime, etc.)
- **get_interfaces:** speed, mac, enabled, description, etc.
- **get_interfaces_counters:** packets, octets, errors
- **get_bgp_neighbors:** AS, IP, received prefixes, accepted prefixes, etc.
- **get_environment:** fan, temp, power, cpu, mem

- **get_lldp_neighbors:** hostname, port

Note: NAPALM requires some prerequisites for properly interacting with the cisco devices. For more information, pls refer the documentation: https://napalm.readthedocs.io/en/latest/support/ios.html

Task 7: Collecting facts from network devices

The python script will

- login to the iosv-1 device via SSH
- collect the device facts
- print the output

```
#!/usr/bin/env python
from napalm import get_network_driver

# Load driver and connection parameters
driver = get_network_driver('ios')
device = driver('198.18.1.55', 'cisco', 'cisco')

# Open connection to the end device and print facts
device.open()
print('NAPALM is running.........\n')
facts = device.get_facts()
print(facts)
device.close()
```

Task 8: Viewing facts with Pretty print

By default, napalm get_facts will provide a structured data output in dictionary format. To make it more readable we will use pprint module.

```
#!/usr/bin/env python
import pprint
from napalm import get_network_driver

# Load driver and connection parameters
driver = get_network_driver('ios')
device = driver('198.18.1.55', 'cisco', 'cisco')

# Open connection to the end device and collect facts
device.open()
print('NAPALM is running.........\n')
facts = device.get_facts()

# Print facts using pretty printer module
pp = pprint.PrettyPrinter(indent=4)
pp.pprint(facts)
device.close()
```

Task 9: Configuring a network device with load_merge option

In this task we configure the ios device with couple of loopback interfaces. The config changes will be provided by the new_loopback.cfg file

new_loopback.cfg

```
!
interface Loopback100
 ip address 1.1.1.100 255.255.255.255
!
interface Loopback200
 ip address 1.1.1.200 255.255.255.255
```

Now we load the new config file using 'load_merge_candidate()' and use 'compare_config()' to compare it with the running config and print the diffs. If in case, there are no changes required the script will print the output "no changes needed".

'+' indicates new config will be added '-' indicates the config will be removed

Rerun the same script and you can verify the output. The script will print “No changes needed”

```
#!/usr/bin/env python
import pprint
from napalm import get_network_driver

# Load driver and connection parameters
driver = get_network_driver('ios')
device = driver('198.18.1.55', 'cisco', 'cisco')

# Open connection to the end device and load config
device.open()
print('Napalm Is Running.........\n')
device.load_merge_candidate(filename='new_loopback.cfg')
diffs = device.compare_config()

# Compare configs to check whether any changes required
# If changes are required commit configs, if not do nothing
if len(diffs) > 0:
    print(diffs)
```

```
    print('Committing changes...')
    device.commit_config()
    print('Done')
else:
    print('No changes needed')

device.close()
```

Task 10: Rollback the device configuration

During change windows there might be instances where we want to rollback the committed changes. In such cases we can use rollback() feature revert the changes back to original state.

```
import pprint
from napalm import get_network_driver

# Load driver and connection parameters
driver = get_network_driver('ios')
device = driver('198.18.1.55', 'cisco', 'cisco')
device.open()
print('Napalm Is Running........\n')

# Rollback the previous config changes
device.rollback()

# Collect facts and print the device details
pp = pprint.PrettyPrinter(indent=4)
facts = device.get_facts()
pp.pprint(facts['interface_list'])

device.close()
```

Task 11: Reset lab config to default state

Finally we will use the reset script to bring back all the devices to its original state.

```
#!/usr/bin/env python
from netmiko import Netmiko

# Sending device ip's stored in a file
with open('device_list') as f:
    device_list = f.read().splitlines()

# Iterate through device list and configure the devices
for device in device_list:
    print ('Connecting to device ' + device)
    ip_address_of_device = device

    # SSH Connection details
    ios_device = {
        'device_type': 'cisco_ios',
        'ip': ip_address_of_device,
        'username': 'cisco',
        'password': 'cisco'
    }

    net_connect = Netmiko(**ios_device)
    output = net_connect.send_config_from_file('reset_config')
    print (output)
```

Conclusion

You have successfully completed the lab. Now you should be able to realize the power of python programming, where you initially started with a very simple python scripts, then quickly modified it to build additional logic and error handling to automate the day to day network admin tasks. This is just an introduction to python, the reference materials will help you become a better programmer and understand all of the programming and automation concepts in detail. Hope we have given you a head start with your Network automation and programmability journey.

Thanks for attending this lab, please share your valuable feedback.

Reference

Additional Learning and Reference materials:

https://developer.cisco.com/startnow/
https://developer.cisco.com/video/net-prog-basics/
https://developer.cisco.com/netdevops/live/
https://pynet.twb-tech.com/blog/automation/netmiko.html
https://napalm.readthedocs.io/en/latest/

Books:
https://www.amazon.com/Network-Programmability-Automation-Next-Generation-Engineer/dp/1491931256

https://www.amazon.com/Python-Crash-Course-2nd-Edition/dp/1593279280