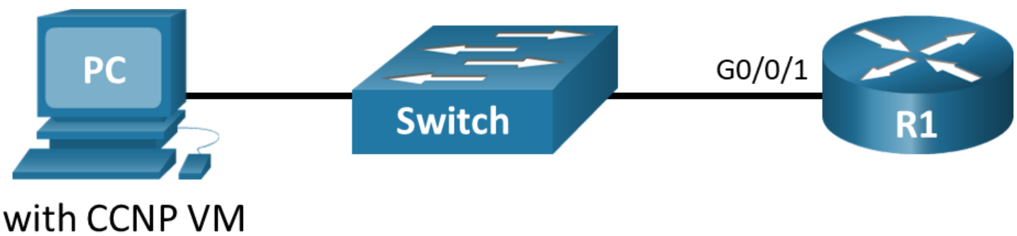
Lab - Use the Netmiko Python Module to Configure a Router (Instructor Version)

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

Answers: 28.1.3 Lab - Use the Netmiko Python Module to Configure a Router

Topology



Addressing Table

Device	Interface	IP Address	Subnet Mask
R1	G0/0/1	192.168.1.1	255.255.255.0
PC	NIC	DHCP	DHCP

Objectives

Part 1: Build the Network and Verify Connectivity

Part 2: Import Netmiko Python Module

Part 3: Use Netmiko to Connect to the SSH Service

Part 4: Use Netmiko to Send Verification Commands

Part 5: Use Netmiko to Send and Verify a Configuration

Part 6: Use Netmiko to Send an Erroneous Command

Part 7: Modify the Program Used in this Lab

Background / Scenario

With the evolution of the Python language, the **netmiko** Python module has emerged as an open source project hosted and maintained on GitHub. In this lab activity, you will use **netmiko** in a Python script to configure and verify a router.

Required Resources

- 1 Router (Cisco 4221 with Cisco IOS XE Release 16.9.4 universal image or comparable)
- 1 Switch (Optional: any switch available for connecting R1 and the PC)

- 1 PC (Choice of operating system with Cisco Networking Academy CCNP VM running in a virtual machine and terminal emulation program)
- Ethernet cables as shown in the topology

Instructions

Part 1: Build the Network and Verify Connectivity

In Part 1, you will cable the devices, start the CCNP VM, and configure R1 for access over an SSH connection. You will then verify connectivity between the CCNP VM and R1, as well as test an SSH connection to R1.

Step 1: Cable the network as shown in the topology.

Connect the devices as shown in the topology diagram.

Step 2: Start the CCNP VM.

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

a. Open VirtualBox and start the **CCNP VM** virtual machine.

b. Enter the password **StudentPass** to access the Ubuntu desktop if necessary

Step 3: Configure R1.

Console into R1 and apply the following configuration to configure basic settings and enable NETCONF, RESTCONF, and SSH.

```
enable
configure terminal
hostname R1
no ip domain lookup
line con 0
logging synchronous
 exec-timeout 0 0
 logging synchronous
line vty 0 15
 exec-t 0 0
 logg sync
 login local
 transport input ssh
ip domain name example.netacad.com
crypto key generate rsa modulus 2048
username cisco priv 15 password cisco123!
interface GigabitEthernet0/0/1
 description Link to PC
 ip address 192.168.1.1 255.255.255.0
 no shutdown
ip dhcp excluded-address 192.168.1.1 192.168.1.10
!Configure a DHCP server to assign IPv4 addressing to the CCNP VM
```

```
ip dhcp pool LAN
 network 192.168.1.0 /24
 default-router 192.168.1.1
 domain-name example.netacad.com
end
copy run start
```

Step 4: Verify the CCNP VM can ping the default gateway.

a. In the **CCNP VM**, open a terminal.

b. Verify the CCNP VM is connected to R1 in a terminal window by either entering **ip address** to verify the CCNP VM received IP addressing from the DHCP server, or simply by pinging R1 at 192.168.1.1. Enter **Ctrl+C** to break out of the ping, as shown in the example output below.

```
student@CCNP:~$ ip address
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens160: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:50:56:b3:72:3b brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.15/24 brd 192.168.1.255 scope global dynamic noprefixroute ens160
       valid_lft 79564sec preferred_lft 79564sec
    inet6 fe80::1ae4:952f:402d:6b1/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: ens192: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:50:56:b3:26:b6 brd ff:ff:ff:ff:ff:ff
    inet 192.168.50.183/24 brd 192.168.50.255 scope global dynamic noprefixroute ens192
       valid_lft 70687sec preferred_lft 70687sec
    inet6 fe80::4c87:a2b3:aa9:5470/64 scope link noprefixroute
       valid_lft forever preferred_lft forever

student@CCNP:~$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=255 time=0.703 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=255 time=0.748 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=255 time=0.757 ms
^C
--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2033ms
rtt min/avg/max/mdev = 0.703/0.736/0.757/0.023 ms
```

c. If the **CCNP VM** has not received IPv4 addressing, check your physical connections between the host PC and R1. Also, verify that R1 is configured correctly according to the previous step.

Step 5: Establish an SSH connection to R1.

a. Open the PuTTY SSH Client.

b. Enter the IPv4 address for the default gateway, 192.168.1.1, and click **Open**.

c. You should be able to login to R1 with the username **cisco** and password **cisco123!**. If not, verify that your SSH configuration is correct on R1.

d. Terminate your SSH session.

Part 2: Import Netmiko Python Module

In Part 2, you will install the **netmiko** module into your Python environment. Netmiko uses an SSH connection to access network devices. It has built in functionality to execute verification commands and apply new commands to the running configuration.

Explore the netmiko module on the project GitHub repository: https://github.com/ktbyers/netmiko.

In the **CCNP VM**, start IDLE and verify that netmiko is installed by importing it as shown.

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> import netmiko
```

Part 3: Use Netmiko to Connect to the SSH Service

In Part 3, you will use netmiko to connect to the SSH service running on R1.

Step 1: Create a new script file.

In IDLE, select **File > New**.

Save the file as **netmiko-script.py**.

Step 2: Create a variable with the SSH attributes.

The netmiko module includes the **ConnectHandler()** function. This function requires the following parameters for establishing an SSH connection with the IOS XE device:

- `device_type` - identifies the remote device type
- `host` - the address (host or IP) of the remote device
- `port` - the remote port of the ssh service
- `username` - remote ssh username
- `password` - remote ssh password

a. In your **netmiko-script.py**, import netmiko's **ConnectHandler()** function.

```
from netmiko import ConnectHandler
```

b. Enter the following information for the **ConnectHandler()** function.

```
sshCli = ConnectHandler(
    device_type = 'cisco_ios',
    host = '192.168.1.1',
    port = 22,
    username = 'cisco',
    password = 'cisco123!'
```

```
)
```

c. Save the script and run it. You will see the following output if your script did not have an error:

```
================== RESTART: /home/student/netmiko-script.py ==================
>>>
```

Part 4: Use Netmiko to Send a Verification Command

In Part 4, you will use your Python script to send verification commands to router R1.

Step 1: Use the netmiko function send_command to send a command through the SSH session.

a. Set a variable to hold the output of the **show** command you are sending. Use the **send_command** function of the **sshCli** object, which is the SSH session previously established, to send the desired command. In this case, we are sending **sh ip int br**. Notice that the command does not have to be the full command. It can be any command that the IOS CLI would accept.

```
output = sshCli.send_command("sh ip int br")
```

b. Run the program. You will see the following output if your script did not have an error:

```
================== RESTART: /home/student/netmiko-script.py ==================
>>>
```

Step 2: Print and format the content of the output variable.

a. The returned content from the function is stored in the **output** variable. In the interactive interpreter, enter **output** to see the content of the variable as shown in the following:

```
================== RESTART: /home/student/netmiko-script.py ==================
>>> output
'Interface              IP-Address      OK? Method Status
Protocol\nGigabitEthernet0/0/0   unassigned      YES unset  administratively down down
\nGigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
\nSerial0/1/0            unassigned      YES unset  administratively down down
\nSerial0/1/1            unassigned      YES unset  administratively down down
\nGigabitEthernet0       unassigned      YES unset  administratively down down    '
```

b. The content of the **output** variable can be made readable with the **format** option of the **print()** command. Also, the **"{}\n."** is used here to add a blank line after the output is printed.

```
print("{}\n".format(output))
```

c. Run your program now and you should get the following result, which is similar to what you would get when you enter the command directly into the IOS CLI.

```
================== RESTART: /home/student/netmiko-script.py ==================
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0/0   unassigned      YES unset  administratively down down
GigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
Serial0/1/0            unassigned      YES unset  administratively down down
Serial0/1/1            unassigned      YES unset  administratively down down
GigabitEthernet0       unassigned      YES unset  administratively down down
```

Part 5: Use Netmiko to Send and Verify a Configuration

In Part 5, you will send configuration commands to create a new loopback interface on R1 and then verify that the interface was created.

Instructor Note: If you are in a lab environment with multiple student completing this lab at the same time, use a scheme to assign students a unique loopback interface and IPv4 address. For example, you could use a table like the following. Add as many loopback addresses as you need.

Student Name	Loopback	IPv4 Address
	10	10.10.1.1/24
	12	10.12.1.1/24
	14	10.14.1.1/24
	16	10.16.1.1/24

Step 1: Create and send a list of configuration commands to R1.

a. Add the following **config_commmands** list variable to your **netmiko-script.py**. If multiple students are accessing R1 at the same time, use the loopback assigned to you by your instructor. Otherwise, you can use loopback 1, as shown below.

   Note: Replace [Student Name] with your name. Leave the \ (backslash) in the **description** command. The backslash escapes the apostrophe so that Python does not read it as a closing quote, but as an apostrophe.

```
config_commands = [
    'int loopback 1',
    'ip add 10.1.1.1 255.255.255.0',
    'description [Student Name]\'s loopback'
    ]
```

b. Create a new variable called **sentConfig** to hold the results. Then use the **send_config_set** function of the **sshCli** object to send the commands to R1.

```
sentConfig = sshCli.send_config_set(config_commands)
```

Step 2: Print and format the content of the sentConfig variable.

a. Use the **print** function to format and display what is stored in **sentConfig**. Then, resend the **send_command** function and repeat the **print** function for the **output** variable so that you can see the new loopback interface in the **show ip interface brief** output.

```
print("{}\n".format(sentConfig))

output = sshCli.send_command("sh ip int br")
print("{}\n".format(output))
```

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

```
================== RESTART: /home/student/netmiko-script.py ==================
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0/0   unassigned      YES unset  administratively down down
GigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
Serial0/1/0            unassigned      YES unset  administratively down down
Serial0/1/1            unassigned      YES unset  administratively down down
GigabitEthernet0       unassigned      YES unset  administratively down down

config term
Enter configuration commands, one per line.  End with CNTL/Z.
```

```
R1(config)#int loopback 1
R1(config-if)#ip add 10.1.1.1 255.255.255.0
R1(config-if)#description [Student Name]'s loopback
R1(config-if)#end
R1#

Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0/0   unassigned      YES unset  administratively down down
GigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
Serial0/1/0            unassigned      YES unset  administratively down down
Serial0/1/1            unassigned      YES unset  administratively down down
GigabitEthernet0       unassigned      YES unset  administratively down down
Loopback1              10.1.1.1        YES manual up                    up
```

Part 6: Use Netmiko to Send an Erroneous Command

Netmiko works similarly to copying and pasting a configuration script into the router CLI. Therefore, like when you paste a script in, netmiko will execute every command that it can. If a command fails, it will continue with the next command. Other automation tools typically will not apply any configuration changes if one or more commands are rejected. In Part 6, you will use netmiko to configure a new loopback address with a duplicate IPv4 address.

Step 1: Create a new loopback interface with the same IPv4 address.

Copy the **config_commands** variable to the bottom of your **netmiko-script.py** script and modify it to store a new loopback interface that uses the same IPv4 address as the previous loopback interface. If multiple students are accessing R1 at the same time, add 1 to the number of the loopback assigned to you by your instructor. Otherwise, you can use loopback 2, as shown below.

```
config_commands = [
    'int loopback 2',
    'ip add 10.1.1.1 255.255.255.0',
    'description [Student Name]\'s loopback'
    ]
```

Step 2: Print and format the content of the sentConfig and output variables.

a. Send the commands and print the output like you did for the first loopback interface.

```
sentConfig = sshCli.send_config_set(config_commands)
print("{}\n".format(sentConfig))

output = sshCli.send_command("sh ip int br")
print("{}\n".format(output))
```

b. Save and run your program. You should get output similar to the output below. Notice that the new loopback interface was configured. However, it is not active because the IPv4 address duplicates the previous loopback interface.

```
================== RESTART: /home/student/netmiko-script.py ==================
(output from Part 4 and Part 5 omitted)

config term
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#int loopback 2
```

```
R1(config-if)#ip add 10.1.1.1 255.255.255.0
% 10.1.1.0 overlaps with Loopback1
R1(config-if)#description [Student Name]'s loopback
R1(config-if)#end
R1#

Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet0/0/0   unassigned      YES unset  administratively down down
GigabitEthernet0/0/1   192.168.1.1     YES manual up                    up
Serial0/1/0            unassigned      YES unset  administratively down down
Serial0/1/1            unassigned      YES unset  administratively down down
GigabitEthernet0       unassigned      YES unset  administratively down down
Loopback1              10.1.1.1        YES manual up                    up
Loopback2              unassigned      YES unset  up                    up
```

Part 7: Modify the Program Used in this Lab

The following is the complete program used in this lab. Practice your Python skills by modifying the program to send different verification and configuration commands.

```
from netmiko import ConnectHandler

sshCli = ConnectHandler(
    device_type = 'cisco_ios',
    host = '192.168.1.1',
    port = 22,
    username = 'cisco',
    password = 'cisco123!'
    )

output = sshCli.send_command("sh ip int br")
print("{}\n".format(output))

config_commands = [
    'int loopback 1',
    'ip add 10.1.1.1 255.255.255.0',
    'description [Student Name]\'s loopback'
    ]

sentConfig = sshCli.send_config_set(config_commands)
print("{}\n".format(sentConfig))

output = sshCli.send_command("sh ip int br")
print("{}\n".format(output))

config_commands = [
    'int loopback 2',
    'ip add 10.1.1.1 255.255.255.0',
```

```
        'description [Student Name]\'s loopback'
        ]

    sentConfig = sshCli.send_config_set(config_commands)
    print("{}\n".format(sentConfig))

    output = sshCli.send_command("sh ip int br")
    print("{}\n".format(output))
```

Router Interface Summary Table

Router Model	Ethernet Interface #1	Ethernet Interface #2	Serial Interface #1	Serial Interface #2
1800	Fast Ethernet 0/0 (F0/0)	Fast Ethernet 0/1 (F0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
1900	Gigabit Ethernet 0/0 (G0/0)	Gigabit Ethernet 0/1 (G0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
2801	Fast Ethernet 0/0 (F0/0)	Fast Ethernet 0/1 (F0/1)	Serial 0/1/0 (S0/1/0)	Serial 0/1/1 (S0/1/1)
2811	Fast Ethernet 0/0 (F0/0)	Fast Ethernet 0/1 (F0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
2900	Gigabit Ethernet 0/0 (G0/0)	Gigabit Ethernet 0/1 (G0/1)	Serial 0/0/0 (S0/0/0)	Serial 0/0/1 (S0/0/1)
4221	Gigabit Ethernet 0/0/0 (G0/0/0)	Gigabit Ethernet 0/0/1 (G0/0/1)	Serial 0/1/0 (S0/1/0)	Serial 0/1/1 (S0/1/1)
4300	Gigabit Ethernet 0/0/0 (G0/0/0)	Gigabit Ethernet 0/0/1 (G0/0/1)	Serial 0/1/0 (S0/1/0)	Serial 0/1/1 (S0/1/1)

Note: To find out how the router is configured, look at the interfaces to identify the type of router and how many interfaces the router has. There is no way to effectively list all the combinations of configurations for each router class. This table includes identifiers for the possible combinations of Ethernet and Serial interfaces in the device. The table does not include any other type of interface, even though a specific router may contain one. An example of this might be an ISDN BRI interface. The string in parenthesis is the legal abbreviation that can be used in Cisco IOS commands to represent the interface.

Device Configs - Final

Router R1

```
R1# show run
Building configuration...


Current configuration : 1687 bytes
!
version 16.9
service timestamps debug datetime msec
service timestamps log datetime msec
```

```
platform qfp utilization monitor load 80
no platform punt-keepalive disable-kernel-core
!
hostname R1
!
boot-start-marker
boot-end-marker
!
no aaa new-model
!
no ip domain lookup
ip domain name example.netacad.com
ip dhcp excluded-address 192.168.1.1 192.168.1.10
!
ip dhcp pool LAN
 network 192.168.1.0 255.255.255.0
 default-router 192.168.1.1
 domain-name example.netacad.com
!
login on-success log
!
subscriber templating
!
multilink bundle-name authenticated
!
diagnostic bootup level minimal
!
spanning-tree extend system-id
!
username cisco privilege 15 password 0 cisco123!
!
redundancy
 mode none
!
interface Loopback1
 ip address 10.1.1.1 255.255.255.0
!
interface Loopback2
 description Student's loopback
 no ip address
!
interface GigabitEthernet0/0/0
 no ip address
 negotiation auto
!
interface GigabitEthernet0/0/1
 description Link to PC
 ip address 192.168.1.1 255.255.255.0
 negotiation auto
!
```

```
interface Serial0/1/0
 no ip address
!
interface Serial0/1/1
 no ip address
!
ip forward-protocol nd
no ip http server
ip http secure-server
!
control-plane
!
line con 0
 exec-timeout 0 0
 logging synchronous
 transport input none
 stopbits 1
line aux 0
 stopbits 1
line vty 0 4
 exec-timeout 0 0
 logging synchronous
 login local
 transport input ssh
line vty 5 15
 exec-timeout 0 0
 logging synchronous
 login local
 transport input ssh
!
end
```