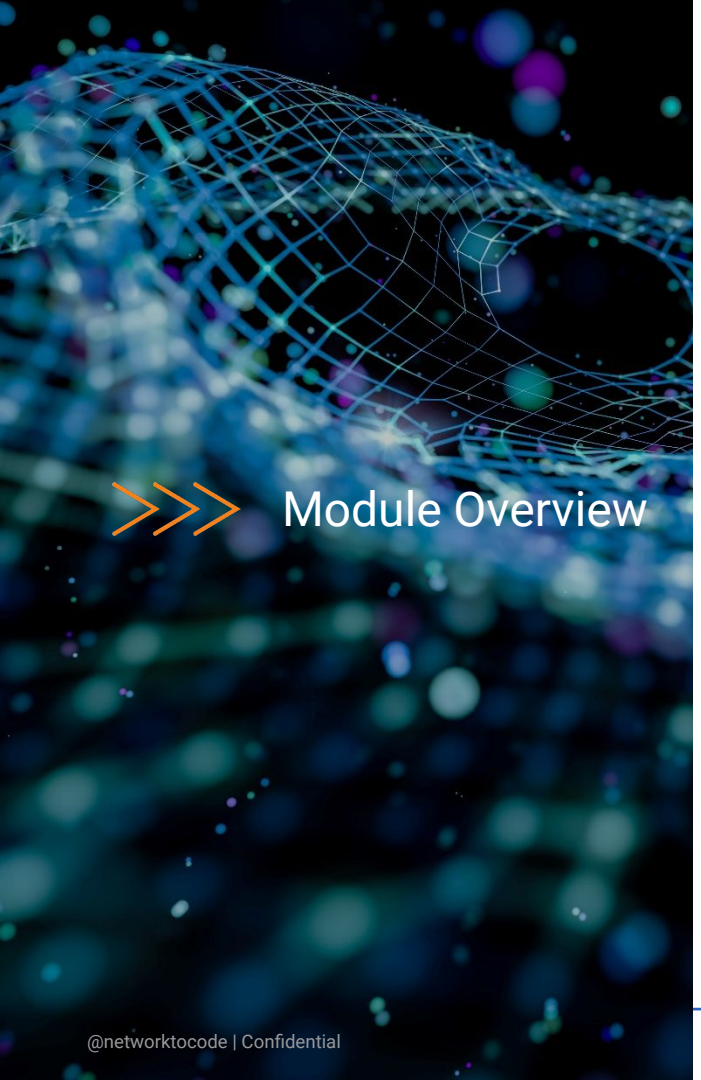
Network Automation with Python

Module 2: Automating Network Devices Using Python



Module Overview

- Python Libraries - netmiko
- Conditionals in Python
- Loops in Python
- Functions in Python
- Command Line Arguments
- Bonus: RegEx, TextFSM, NAPALM

Module Table of Contents

Lecture 5: Python Network Libraries

Topic 12 - Introduction to Netmiko

Labs 12-13

Topic 12: Introduction to Netmiko

Netmiko overview and supported network device platforms

Using Netmiko to run commands remotely on network devices

>>> Third-party Packages in Python

- The Python “standard library” is what comes installed with the Python interpreter by default
- In managing a Python project, it is common to install other Python packages providing additional functionality to your Python programs or even standalone CLI commands.
  - Examples: requests, netmiko, napalm, ansible, salt, black, pylint etc.
- Python Packages
  - Installed using OS package manager (uncommon) - e.g. `apt install python3-pip python3-yaml`
  - Installed using the Python package installer **pip** which pulls packages from the Python Package Index (https://pypi.org/)
  - Installed using any other Python project management tools like poetry, pipenv etc.
- Where does Python find packages?
  - In its installation paths, which are OS dependent
  - In its PYTHONPATH, which is a list of additional paths where it might find code
    - Check its value by printing sys.path from the Python interpreter

```
>>> import sys
>>> print(sys.path)
['', '/usr/local/lib/python38.zip', '/usr/local/lib/python3.8', '/usr/local/lib/python3.8/lib-dynload',
'/usr/local/lib/python3.8/site-packages']
```

>>> Netmiko Overview

The netmiko package is:

- A Python library that simplifies SSH management to network devices
- Based on the Paramiko SSH library

The purposes of the library are the following:

- 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 including possibly commit actions.
- Do the above across a broad set of networking vendors and platforms.

>>> Supported Platforms

Netmiko - regularly tested (and widely used)

- Arista vEOS
- Cisco ASA
- Cisco IOS
- Cisco IOS-XE
- Cisco IOS-XR
- Cisco NX-OS
- Cisco SG300
- HP ProCurve
- Juniper Junos
- Linux

Other platforms with limited testing

- 6Wind
- Adtran
- Dell
- Extreme
- HPE
- Huawei
- Mellanox
- MikroTik
- Ruckus
- Vyatta
- ZTE

Full list of platforms can be found here

>>> Getting Started with Netmiko

Starting a connection to the device by creating an object that maintains the ssh session:

```
>>> from netmiko import ConnectHandler
>>>
>>> device = ConnectHandler(device_type='cisco_nxos', ip='nxos-spine1', username='ntc', password='ntc123')
>>>
```

>>> Getting Started with Netmiko

Starting a connection to the device by creating an object that maintains the ssh session:

```
>>> from netmiko import ConnectHandler
>>>
>>> device = ConnectHandler(device_type='cisco_nxos', ip='nxos-spine1', username='ntc', password='ntc123')
>>>
```

Note: You can simply "`import netmiko`" but then you must type "`device = netmiko.ConnectHandler(...)`"

Send a command to the device

```
>>> device.send_command('show hostname')
u'nxos-spine1 '
>>>
```

>>> Using Netmiko

In the interactive prompt, the ssh session may time out, verify if connection is live with the `is_alive()` method which returns True/False.

```
>>> print("Connecting to CSR1...")
Connecting to CSR1...
>>> csr1 = ConnectHandler(ip='csr1', username=user, password=pwd, device_type=d_type)
>>> print("Connected to CSR1")
Connected to CSR1
>>> csr1_device_check = csr1.is_alive()
>>> print("Connection to device is verified " + str(csr1_device_check))
Connection to device is verified True
>>>
```

To reconnect if the session is lost use the `.establish_connection()` method

```
>>> csr1.disconnect()
>>> csr1.is_alive()
False
>>> csr1.establish_connection()

u''
>>> csr1.is_alive()
True
>>>
```

>>> Using netmiko (cont'd)

Issuing Configuration changes, use `send_config_set` with a list of commands. Netmiko will automatically enter and exit config mode

```
>>> commands = ['interface Loopback100', 'ip address 10.200.1.20 255.255.255.0']
>>> output = csr1.send_config_set(commands)

>>> print(output)
config term
Enter configuration commands, one per line.  End with CNTL/Z.
csr1(config)#interface Loopback100
csr1(config-if)#ip address 10.200.1.20 255.255.255.0
csr1(config-if)#end
csr1#
>>>
```

>>> Using netmiko (cont'd)

Or you can send configuration commands from a file

```
>>> csr1 = ConnectHandler(host='csr1', username='ntc', password='ntc123', device_type='cisco_ios')
>>> csr2 = ConnectHandler(host='csr2', username='ntc', password='ntc123', device_type='cisco_ios')

>>> csr1.send_config_from_file("snmp_communities.txt")
u'config term\nEnter configuration commands, one per line.  End with
CNTL/Z.\ncsr1(config)#snmp-server community networktocode ro\ncsr1(config)#snmp-server community
public ro\ncsr1(config)#snmp-server community private rw\ncsr1(config)#snmp-server community
supersecret rw\ncsr1(config)#end\ncsr1#'

>>> csr2.send_config_from_file("snmp_communities.txt")
u'config term\nEnter configuration commands, one per line.  End with
CNTL/Z.\ncsr2(config)#snmp-server community networktocode ro\ncsr2(config)#snmp-server community
public ro\ncsr2(config)#snmp-server community private rw\ncsr2(config)#snmp-server community
supersecret rw\ncsr2(config)#end\ncsr2#'
```

>>> Using netmiko (cont'd)

Storing & Printing a command response

```
>>> vlans = device.send_command('show vlans')
>>>
>>> print(vlans)

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Eth1/2, Eth1/3, Eth1/8, Eth1/9
                                                Eth1/11, Eth1/12, Eth1/13, Eth2/9
                                                Eth2/10, Eth2/11, Eth2/12
2    VLAN0002                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
3    VLAN0003                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
4    VLAN0004                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
5    VLAN0005                         active    Po10, Po11, Po12, Eth1/4
                                                Eth1/5, Eth1/6, Eth1/7, Eth2/5
                                                Eth2/6
# shortened for brevity
```

>>> Primary List of Methods

- config_mode() – Enter into config mode
- enable() – Enter enable mode
- establish_connection() – Establish SSH connection to device
- exit_enable_mode() – Exit enable mode
- find_prompt() – Return the current router prompt
- commit() – Execute a commit action on Juniper and IOS-XR
- disconnect() – Close the SSH connection
- send_command_timing() - Send command down the SSH channel, return output back (uses timer to wait for device)
- send_command_expect() – Send command to device; retrieve output until router_prompt or expect_string
- send_config_set() – Send a set of configuration commands to remote device
- send_config_from_file() – Send a set of configuration commands loaded from a file

>>> Summary

- Legacy devices are here to stay (for awhile)
- Great way to bridge the gap between legacy and modern devices that return structured data
- Netmiko is a great library to integrate with TextFSM to create a pseudo-API
  - CLI commands gets sent to the device and you get returned structured data
  - We cover how to do this with Ansible

Lecture 5: Python Network Libraries

Labs 12-13

>>> Lab Time

- Lab 12 - Getting Started with Netmiko
- Lab 13 - Challenge - Use Netmiko to Create Configuration Backups
  - Challenge means you should try to write the script on your own first!
  - The lab provides the solution so you can peek ahead if you get stuck (or ask your instructor!)

Lecture 6: Control Structures in Python

Topic 13 - Conditionals

Topic 14 - Loops

Labs 14-16

Topic 13: Conditionals

Understand the If Statement and its branching syntax

Nested Conditionals

>>> if statement

- Colon at the end of the statement is required
- 4 spaces for indentation
- Uses operators for boolean comparisons
- Task:
  - Check to see if a device is running a particular version of software. If it is, store the hostname of that device into a list.

```
>>> facts
{'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'hostname': 'NYC301', 'os': '6.1.2'}
```

```
>>> devices = []
>>> if facts['os'] == '6.1.2':
...     devices.append(facts['hostname'])
...
```

```
>>> print(devices)
['NYC301']
```

>>> if Examples

- Use operators such as ==, !=, >, <

```
>>> hostname = 'R1'
>>>
>>> if hostname.lower() == 'r1':
...     print("Hostname Correct")
...
Hostname Correct
>>>
```

```
>>> if hostname.lower() != 'r1':
...     print("Hostname NOT Correct")
...
>>>
```

NOTE SPACING - any amount of spacing, but most important is consistency. **Good practice is 4 spaces.**

```
>>> if 5 > 3:
...  print("Yes, 5 is greater than 3")
...
Yes, 5 is greater than 3
>>>
```

>>> if-else

- Colon at the end of each conditional
- 4 spaces for each conditional block
- Indentation levels must match
- Note:
  - $VAR evaluates to True if the value is not null (none/empty string, list, dictionary, 0), else it is False

```
>>> commands = ''
>>>
>>> if commands:
...     print('Commands to send: ' + commands)
... else:
...     print('No commands to send.')
...
No commands to send.
```

>>> if-elif-else

- Once a condition is met (true/false), the conditional block is exited
- If none are met, the else statement is executed
- Example:
  - Check to find type of a specified interface

```
>>> interface = 'Ethernet1/1'
>>>
>>> if interface.lower().startswith('et'):
...     itype = 'ethernet'
... elif interface.lower().startswith('vl'):
...     itype = 'svi'
... elif interface.lower().startswith('lo'):
...     itype = 'loopback'
... elif interface.lower().startswith('po'):
...     itype = 'portchannel'
... elif interface.lower().startswith('mgmt'):
...     itype = 'management'
... else:
...     itype = 'unknown'
>>>
>>> print(itype)
ethernet
```

>>> if-elif-else

```
>>> interface = 'port-channel20'
>>>
>>> if interface.lower().startswith('et'):
...     itype = 'ethernet'
... elif interface.lower().startswith('vl'):
...     itype = 'svi'
... elif interface.lower().startswith('lo'):
...     itype = 'loopback'
... elif interface.lower().startswith('po'):
...     itype = 'portchannel'
... elif interface.lower().startswith('mgmt'):
...     itype = 'management'
... else:
...     itype = 'unknown'
>>>
>>> print(itype)
portchannel
```

Note: Once the type is known, further statistics or analysis can be performed that are specific to that type of interface.

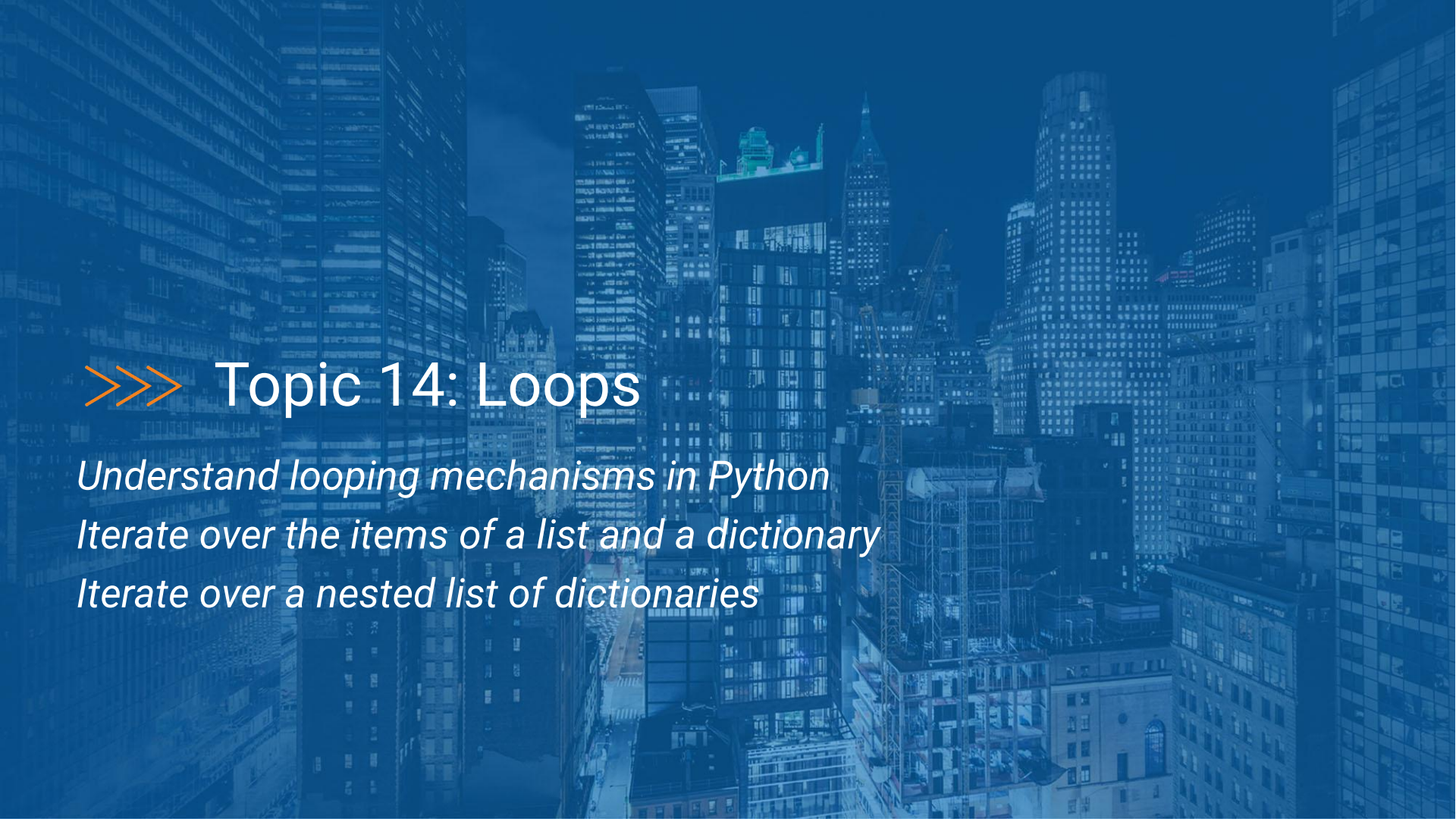
>>> Nested Conditionals

Be very careful with indentation

```
>>> vendor = 'cisco'
>>> platform = 'nexus'
>>> model = '9000'
>>>
>>> if vendor == "cisco":
...     if platform == "nexus":
...         if model == "9000":
...             print("Vendor:", vendor)
...             print("Platform:", platform)
...             print("Model:", model)
...         else:
...             print("unknown model")
...     else:
...         print("unknown platform")
... else:
...     print("unknown vendor")
...
Vendor: cisco
Platform: nexus
Model: 9000
>>>
```

>>> Summary

- Conditionals are logical
- Start with basic logic:
  - if this, do that
  - if this, do that, else do something else...
- Syntax - spacing (4 spaces) and colon on each line with an if, elif, or else
- Remember if the variable is empty, it evaluates to False
  - empty_list = []
  - empty_string = ''
  - empty_dict = {}



Topic 14: Loops

Understand looping mechanisms in Python

Iterate over the items of a list and a dictionary

Iterate over a nested list of dictionaries

>>> for loop

- Iterate through a given set of objects
- User-defined variable in for loop

```
>>> routers = ['r1', 'r2', 'r3']
>>>
>>> for rtr in routers:
...   print(rtr)
...
r1
r2
r3
>>>
```

Common to use item

```
>>> for item in routers:
...   print(item)
...
r1
r2
r3
>>>
```

>>> for loop

- Looping through a list and performing a given operation for each element in the list.

```
>>> interfaces = ['Eth1/1', 'vlan20', 'Eth4/4', 'loop10']
>>>
>>> for interface in interfaces:
...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     elif interface.lower().startswith('po'):
...         itype = 'portchannel'
...     elif interface.lower().startswith('mgmt'):
...         itype = 'management'
...     print(itype)
...
ethernet
svi
ethernet
loopback
>>>
```

>>> Examples

Determine if IPs are rogue within a network when you're tracking all IP addresses and have a working list of "rogue" IP addresses.

```
>>> ips = ['1.1.1.1', '10.1.1.1', '192.168.1.1', '5.5.5.5', '10.1.2.1']
>>> rogue_ips = ['9.9.9.9', '1.1.1.1', '5.5.5.5']
>>>
>>> for ip in ips:
...     if ip in rogue_ips:
...         print(f'Found ROGUE IP: {ip}')
...     else:
...         print('Valid IP:', ip)
...
Found ROGUE IP: 1.1.1.1
Valid IP: 10.1.1.1
Valid IP: 192.168.1.1
Found ROGUE IP: 5.5.5.5
Valid IP: 10.1.2.1
>>>
```

>>> Aside: Dictionary items() Method

- Commonly used with for loops to iterate over keys and values together
- items() simplifies accessing key/value pairs
- Returns a list of tuples and each tuple is two elements
  - Element 1 is the key and Element 2 is value
  - For now, think of tuples as a list, thus the returned object would be a list of lists

```
>>> facts
{'vendor': 'arista', 'mgmt_ip': '10.1.1.1', 'chipset': 't2', 'hostname': 'NYC301', 'os': '6.1.2'}
>>> f_list = facts.items()
>>>
>>> f_list
[('chipset', 't2'), ('hostname', 'NYC301'), ('vendor', 'arista'), ('os', '6.1.2'), ('mgmt_ip',
'10.1.1.1')]
>>>
>>> len(f_list)
5
>>>
>>> f_list[0][0]
'chipset'
>>> f_list[0][1]
't2'
```

>>> Examples

- Looping over all items in a dictionary
- Remember dict.items() returns a list of tuples; each tuple has two elements.
  - The first element in the tuple is the key and the second is the value.

```
>>> facts = {'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista', 'os': '6.1.2',
'mgmt_ip': '10.1.1.1'}
>>>
>>> for key, value in facts.items():
...     print(key, value)
...
chipset t2
hostname NYC301
vendor arista
os 6.1.2
mgmt_ip 10.1.1.1
>>>
```

>>> Examples

- Looping over all items in a dictionary
- Remember dict.items() returns a list of tuples; each tuple has two elements.
  - The first element in the tuple is the fact name (key) and the second is the fact itself (value).

```
>>> facts = {'chipset': 't2', 'hostname': 'NYC301', 'vendor': 'arista', 'os': '6.1.2', 'mgmt_ip':
'10.1.1.1'}
>>>
>>> for device_fact, device_value in facts.items():
...     print(device_fact, device_value)
...
chipset t2
hostname NYC301
vendor arista
os 6.1.2
mgmt_ip 10.1.1.1
>>>
```

>>> Examples

- Looping over list of dictionaries

```
>>> vlans = [{'id': '10', 'name': 'web'}, {'id': '20',
'name': 'app'}, {'id': '30', 'name': 'db'}]
>>>
>>>
>>> for vlan in vlans:
...     print('ID:', vlan['id'])
...     print('NAME:', vlan['name'])
...
ID: 10
NAME: web
ID: 20
NAME: app
ID: 30
NAME: db
>>>
```

```
>>> import json
>>> print(json.dumps(vlans, indent=4))
[
    {
        "id": "10",
        "name": "web"
    },
    {
        "id": "20",
        "name": "app"
    },
    {
        "id": "30",
        "name": "db"
    }
]
```

>>> Summary

- The for loop is an integral part of Python, especially for Network Engineers
  - "for each element in the list, do this"
  - "for each key/value pair in the dictionary, do this"
- Syntax:
  - Colon & indentation
- When getting started…
  - Start small with a single if/elif and then re-factor to using loops

Lecture 6: Control Structures in Python

Labs 14-16

>>> Lab Time

- Lab 14 - Using Conditionals
- Lab 15 - Getting Started with For Loops
  - You will access and print elements as you loop through lists and dictionaries
  - You will iterate through key-value pairs that are commands in the form of feature/command, and using the feature name (command key), you will access their configuration values values from another dictionary, to build a list of commands to send to a network device.
- Lab 16 - Refactoring Code using Loops

Lecture 7: Functions

Topic 15 - Getting Started with Functions

Labs 17-18

Topic 15: Getting Started with Functions

Understanding the syntax in Python and Creating a Function

Passing in arguments and returning data

Function calls within a for loop

>>> Functions

- Reusable and repeatable code should get placed into a function
- Minimize the amount of duplicate statements in your code
- Built-in function called **len**

```
>>> hostname = 'DCNJSWITCH1'
>>>
>>> len(hostname)
11
```

```
>>> commands = ['interface Ethernet1/1', 'switchport access vlan 10']
>>>
>>> len(commands)
2
>>>
```

>>> Creating a Function

- Similar syntax to the for loop (colon and indentation)
  - Spaces, indentation, colon
- Optionally pass in arguments and return data

```
>>> def print_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...     print(vlans)
...
>>>
```

>>> Creating a Function

- Similar syntax to the for loop (colon and indentation)
  - Spaces, indentation, colon
- Optionally pass in arguments and return data

```
>>> def print_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...     print(vlans)
...
>>>
```

```
>>> print_vlans()
[1, 5, 10, 11, 14, 15]
>>>
```

>>> Returning Data from a Function

- Use the return statement to return an object/variable

```
>>> def print_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...     print(vlans)
...
>>>
```

>>> Returning Data from a Function

- Use the return statement to return an object/variable

```
>>> def print_vlans():
...     vlans = [1, 5, 10, 11, 14, 15]
...     print(vlans)
...
>>>
```

```
>>> vlans_list = print_vlans()
>>>
>>> print(vlans_list)
[1, 5, 10, 11, 14, 15]
>>>
```

>>> Returning Data from a Function (cont'd)

- Pass in variable, uses a conditional, and returns an object

```
>>> def get_vlans(device):
...     if device == 'R1':
...         vlans = [1, 5, 10, 11, 14, 15]
...     elif device == 'R2':
...         vlans = [5, 10, 14, 15]
...
...     return vlans
...
>>>
```

>>> Returning Data from a Function (cont'd)

- Pass in variable, uses a conditional, and returns an object

```
>>> def get_vlans(device):
...     if device == 'R1':
...         vlans = [1, 5, 10, 11, 14, 15]
...     elif device == 'R2':
...         vlans = [5, 10, 14, 15]
...
...     return vlans
...
>>>
```

```
>>> vlans = get_vlans('R1')
>>>
>>> print(vlans)
[1, 5, 10, 11, 14, 15]
>>>
```

>>> Passing in Arguments and Returning Data

- Define parameters being passed in the function definition

```
>>> def vlan_exists(vlan_id):
...     vlans = [1, 5, 10, 11, 14, 15]
...     if vlan_id in vlans:
...         return True
...     return False
...
>>>
```

>>> Passing in Arguments and Returning Data

- Define parameters being passed in the function definition

```
>>> def vlan_exists(vlan_id):
...     vlans = [1, 5, 10, 11, 14, 15]
...     if vlan_id in vlans:
...         return True
...     return False
...
>>>
```

```
>>> vlan_exists(10)
True
>>>
>>> if vlan_exists(25):    # you don't have to save the returned data into a var
...     print("Vlan exists")
... else:
...     print("Vlan doesn't exist")
...
Vlan doesn't exist
>>>
```

>>> Function - Example

- Same block of code previously used
- Encapsulated in a function

```
>>> def get_interface_type(interface):
...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('po'):
...         itype = 'portchannel'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     else:
...         itype = 'unknown'
...
...     return itype
...
>>>
```

>>> Function - Example

- Same block of code previously used
- Encapsulated in a function

```
>>> def get_interface_type(interface):
...     if interface.lower().startswith('et'):
...         itype = 'ethernet'
...     elif interface.lower().startswith('vl'):
...         itype = 'svi'
...     elif interface.lower().startswith('po'):
...         itype = 'portchannel'
...     elif interface.lower().startswith('lo'):
...         itype = 'loopback'
...     else:
...         itype = 'unknown'
...
...     return itype
...
>>>
```

```
>>> print(get_interface_type('Ethernet1/48'))
Ethernet
>>> interface_type = get_interface_type('Ethernet1/48')
>>> interface_type
'ethernet'
```

>>> Function Calls within a for loop

- Do you really want to repeat the code in the function numerous times?
- Modularize code and separate logic

```
>>> interfaces = ['Ethernet1/1', 'loopback20', 'mgmt0']
>>>
>>> for interface in interfaces:
...     print(interface)
...     print(get_interface_type(interface))
...     print('--------')
...
Ethernet1/1
ethernet
--------
loopback20
loopback
--------
mgmt0
unknown
--------
```

>>> Passing in Multiple Arguments

These are positional and required arguments

```
>>> def verify_vlan(vlan_id, vlan_name):
...     print(f'The VLAN ID is {vlan_id} while the VLAN Name is {vlan_name}.')
...
>>>
>>> verify_vlan(5, 'web_vlan')
The VLAN ID is 5 while the VLAN Name is web_vlan.
>>>
```

>>> Optional Arguments

Using positional (required) arguments and optional arguments with default values.

```
>>> def interface_settings(interface, speed='auto', duplex='auto'):
...     print("Interface: ", interface)
...     print("Speed:", speed)
...     print("Duplex:", duplex)
...
>>>
```

>>> Optional Arguments

Using positional (required) arguments and optional arguments with default values.

```
>>> def interface_settings(interface, speed='auto', duplex='auto'):
...     print("Interface: ", interface)
...     print("Speed:", speed)
...     print("Duplex:", duplex)
...
>>>
```

```
>>> interface_settings('Eth1')
Interface:  Eth1
Speed: auto
Duplex: auto
>>>
```

```
>>> interface_settings('Eth2', '1000')
Interface:  Eth2
Speed: 1000
Duplex: auto
>>>
```

>>> Bonus: *args and **kwargs

```
#! /usr/bin/env python
# python multiargs.py
def foo(required, required2, *args, **kwargs):
    print("Required arguments are: " + required + " & ")
    print(str(required2))
    if args:
            print(f"Optional args are: { args }")
    if kwargs:
            print(f"Optional keyword args are: { kwargs }")
```

```
>>> foo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() missing 2 required positional arguments: 'required' and 'required2'
>>>
>>> foo("This is required!", "This too")
Required arguments are: This is required! &
This too
```

>>> Bonus: *args and **kwargs

```
>>> foo("This is required!", "This too", 1, 2, "3")
Required arguments are: This is required! &
This too
Optional args are: (1, 2, '3')
>>>
>>> foo("This is required!", "This too", 1, 2, "3", username='ntc')
Required arguments are: This is required! &
This too
Optional args are: (1, 2, '3')
Optional keyword args are: {'username': 'ntc'}
```

>>> Bonus: *args and **kwargs

```
>>> foo("This is required!", "This too", username='ntc')
Required arguments are: This is required! &
This too
Optional keyword args are: {'username': 'ntc'}
>>>
>>> foo("This is required!", "This too", username='ntc', ... device_type= 'cisco_ios')
Required arguments are: This is required! &
This too
Optional keyword args are: {'username': 'ntc', 'device_type': 'cisco_ios'}
```

>>> Bonus: *args and **kwargs

So how can **kwargs be applied to **netmiko**?

```
>>> from netmiko import ConnectHandler
>>> csr_vars = { 'ip': 'csr2', 'device_type': 'cisco_ios',
... 'username': 'ntc', 'password': 'ntc123' }
>>> csr2 = ConnectHandler(**csr_vars)
>>> csr2.is_alive()
True
>>> csr2.send_command('show hostname')
csr2
```

Question: how can you quickly repeat this process for csr3?

```
>>> csr_vars['ip'] = 'csr3'
>>> csr3 = ConnectHandler(**csr_vars)
>>> csr3.is_alive()
True
>>> csr3.send_command('show hostname')
csr3
```

>>> Summary

- Functions provide for a means to reuse code
- From repeatedly entering the same CLI commands to writing reusable functions
- Write a piece of code once, maybe twice, but never more!
  - Opposite of what network engineers do today

Lecture 7: Functions

Labs 17-18

>>> Lab Time

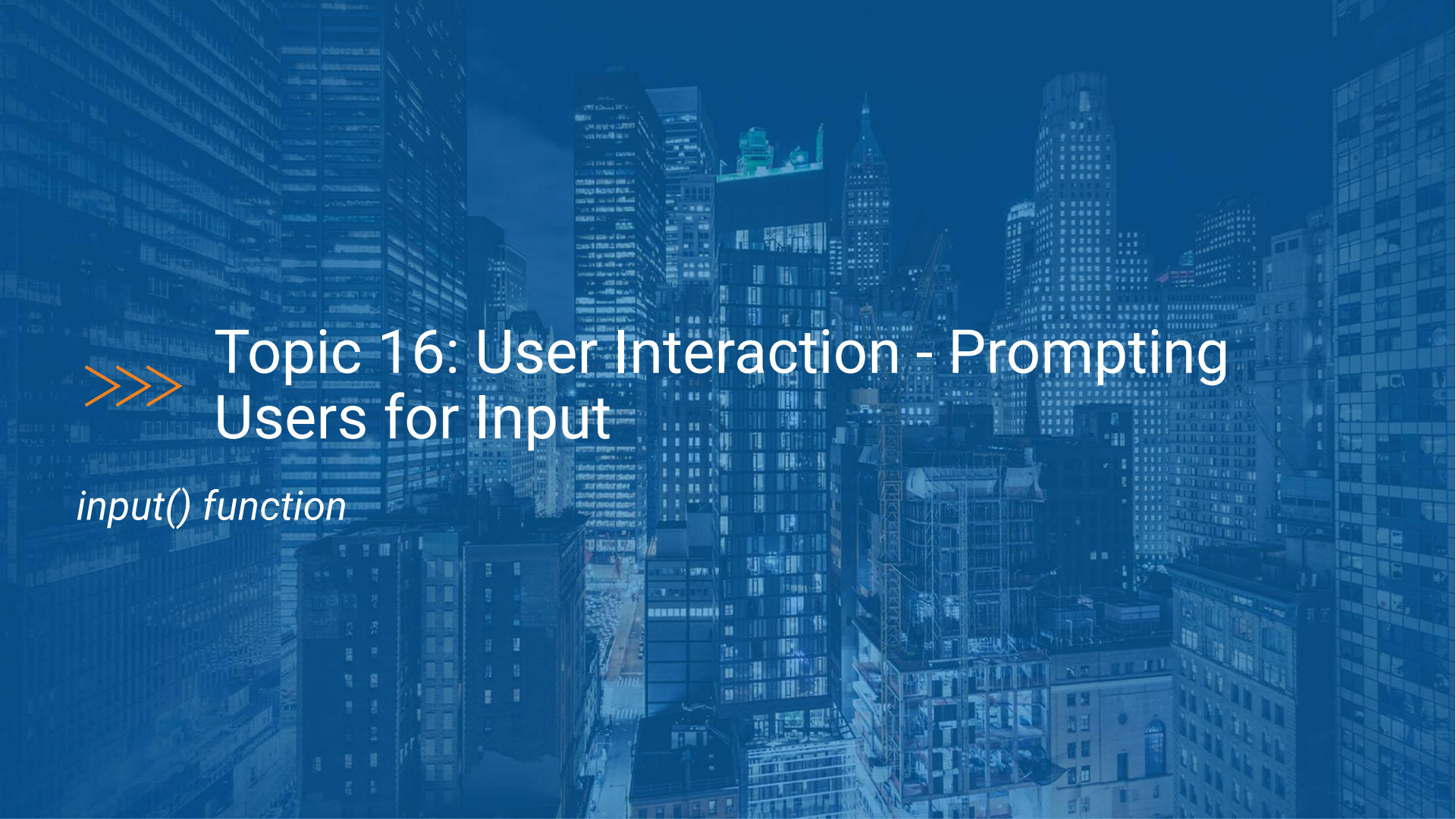
- Lab 17 - Getting Started with Functions
- Lab 18 - Refactoring Code with Functions

Lecture 8: Developing Interactive Command Line Tools

Topic 16 - User Interaction - Prompting Users for Input

Topic 17 - Passing in Arguments

Lab 19



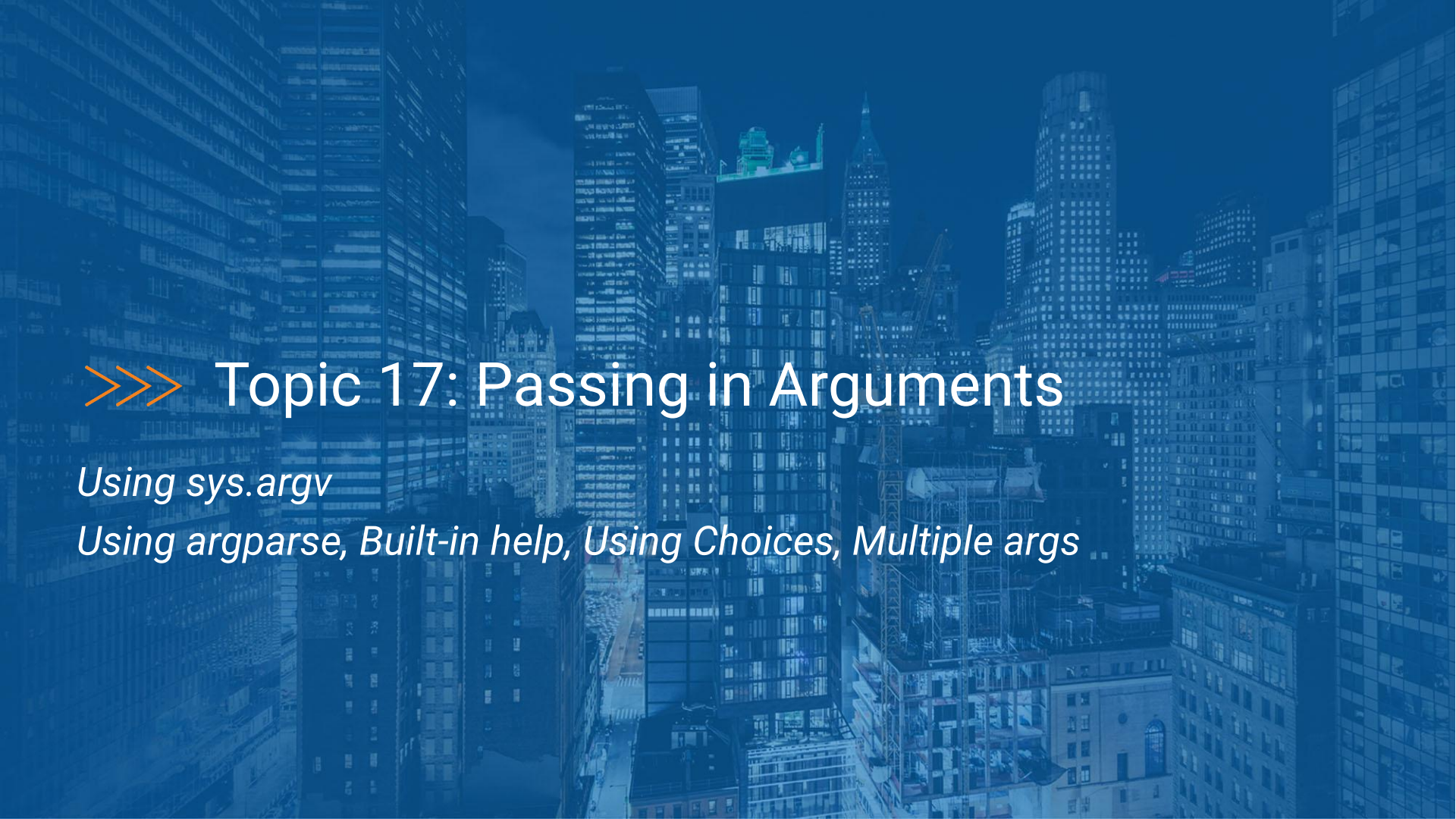
Topic 16: User Interaction - Prompting Users for Input

input() function

>>> User interaction - Prompting users for input

- The input (Python 3.x) built-in function is used to collect user input, interactively
- Prompt the user for data that can then be stored as a variable and used in the script

```
>>> number_of_routers = input('Enter the number of routers in the mesh: ')
Enter the number of routers in the mesh: 6
>>> num_routers = int(number_of_routers)
>>> num_of_conn = int(( num_routers * (num_routers - 1) ) / 2 )
>>> print(f"For a full mesh of {num_routers} routers, you will need {num_of_conn} connections.")
For a full mesh of 6 routers, you will need 15 connections.
```



Topic 17: Passing in Arguments

Using sys.argv

Using argparse, Built-in help, Using Choices, Multiple args

>>> Passing in Arguments

- Using the sys module
  - argv is an attribute (variable) within the sys module that makes it fast and easy to pass variables in from the command line
- Using argparse module
  - Built-in module that allows for more functionality such as defining a help menu and using user-friendly flags

>>> sys.argv

- It's a variable that is of type list

```
#! /usr/bin/env python
import sys

print(sys.argv)
```

- Script name is argv[0]

```
ntc@jump-host:~$ python args_test.py hello world 10.1.1.1 NYCR1
['args_test.py', 'hello', 'world', '10.1.1.1', 'NYCR1']
```

>>> Example - sys.argv

Objective:

- Pass in the 'fact' you want to see the value for and the proper key-value pair will be printed from the facts dictionary.

Dictionary:

```
facts = {'vendor': 'cisco', 'mgmt_ip': '10.1.1.1', 'model': 'nexus', 'hostname': 'NYC301', 'os': '6.1.2'}
```

User experience:

```
ntc@jump-host:~$ python print_facts.py model
model: nexus
```

>>> Examining the Code

```
#! /usr/bin/env python
import sys

facts = {'vendor': 'cisco', 'mgmt_ip': '10.1.1.1', 'model': 'nexus', 'hostname': 'NYC301', 'os': '6.1.2'}
args = sys.argv              # assign argv to args (optional; cleans up the code)
fact_to_print = args[1]      # assign the second element to my_fact
print(fact_to_print + ': ' + facts[fact_to_print])
```

```
ntc@jump-host:~$ python print_facts.py model
model: nexus
```

>>> argparse

- Python module that simplifies defining a help menu, using user-friendly flags, and much more

```
ntc@jump-host:~$ python get_facts.py -f model
model: nexus
ntc@jump-host:~$ python get_facts.py -f=model
model: nexus
ntc@jump-host:~$ python get_facts.py --f=model
model: nexus
ntc@jump-host:~$ python get_facts.py --fact=model
model: nexus
ntc@jump-host:~$ python get_facts.py --fact model
model: nexus
```

```
import argparse


facts = {'vendor': 'cisco', 'mgmt_ip': '10.1.1.1', 'model': 'nexus', 'hostname': 'NYC301', 'os': '6.1.2'}
parser = argparse.ArgumentParser(description='Python Argparse Demo')
parser.add_argument('-f', '--fact', help='enter a valid fact from the device facts dictionary')
args = parser.parse_args()
print(args.fact + ': ' + facts[args.fact])
```

>>> argparse - Built-in help

- Leverage help menu natively built-in
- Can be disabled if needed when parser is instantiated

```
ntc@jump-host:~$ python get_facts.py --help
usage: get_facts.py [-h] [-f FACT]

Python Argparse demo.

optional arguments:
  -h, --help            show this help message and exit
  -f FACT, --fact FACT  enter a valid fact from the device facts dictionary
```

>>> argparse - Choices

- Built-in error validation
- What if the user enters an invalid value for argument?

```
ntc@jump-host:~$ python get_facts.py --f platform
Traceback (most recent call last):
  File "get_facts.py", line 14, in <module>
    print(args.fact + ': ' + facts[args.fact])
KeyError: 'platform'
```

- Use the choices parameter:

```
parser.add_argument('-f', '--fact', choices=['vendor', 'mgmt_ip', 'model', 'hostname', 'os'],
help='enter a valid fact from the device facts dictionary')
```

>>> argparse - Using choices

```
ntc@jump-host:~$ python get_facts.py --f platform
usage: get_facts.py [-h] [-f {vendor,mgmt_ip,model,hostname,os}]
get_facts.py: error: argument -f/--fact: invalid choice: 'platform' (choose from 'vendor', 'mgmt_ip',
'model', 'hostname', 'os')
```

>>> argparse - Using choices

```
ntc@jump-host:~$ python get_facts.py --f platform
usage: get_facts.py [-h] [-f {vendor,mgmt_ip,model,hostname,os}]
get_facts.py: error: argument -f/--fact: invalid choice: 'platform' (choose from 'vendor', 'mgmt_ip',
'model', 'hostname', 'os')
```

```
ntc@jump-host:~$ python get_facts.py -h
usage: get_facts.py [-h] [-f {vendor,mgmt_ip,model,hostname,os}]
Python Argparse Demo
optional arguments:
 -h, --help            show this help message and exit
 -f {vendor,mgmt_ip,model,hostname,os}, --fact {vendor,mgmt_ip,model,hostname,os}
                       enter a valid fact from the device facts dictionary
```

>>> argparse - Multiple arguments

Objective:

- Pass in a fact you want to see the value for, but also include a description
- Code was modified to also print the description

```
ntc@jump-host:~$ python get_facts.py -h
usage: get_facts.py [-h] [-f {vendor,mgmt_ip,model,hostname,os}] [-d DESCR]

Python Argparse Demo

optional arguments:
  -h, --help            show this help message and exit
  -f {vendor,mgmt_ip,model,hostname,os}, --fact {vendor,mgmt_ip,model,hostname,os}
                        enter a valid fact from the device facts dictionary
  -d DESCR, --descr DESCR
                        enter a description for this job
```

```
ntc@jump-host:~$ python get_facts.py -f hostname -d "Test Job"
hostname: NYC301
Test Job
```

>>> argparse - Adding descr argument

```
import argparse

if __name__ == "__main__":
    facts = {'vendor': 'cisco', 'mgmt_ip': '10.1.1.1', 'model': 'nexus', 'hostname': 'NYC301', 'os':
'6.1.2'}

    parser = argparse.ArgumentParser(description='Python Argparse Demo')
    parser.add_argument('-f', '--fact', help='enter a valid fact from the device facts dictionary')
    parser.add_argument('-d', '--descr', help='enter a description for this job')

    args = parser.parse_args()

    print(args.fact + ': ' + facts[args.fact])
    print(args.descr)
```

>>> Summary

- For quick testing **sys.argv** is a great option
- For a more robust script, you want others to use and to have a more defined help menu, **argparse** is the way to go
  - Supports more features, feel free to continue to digging!

Lecture 8: Developing Interactive Command Line Tools

Lab 19

>>> Lab Time

- Lab 19 - Passing in User Input
  - Prompt user input using input and process the input
  - Continue to build on the neighbors script from previous labs and only print certain neighbor and device information based on the arguments being passed in
  - Write a basic script using sys.argv that prints arguments


>>> network .toCode()
Python Libraries - Bonus Topics

Regular Expressions

RegEx Overview

>>> RegEx Overview

- A Regular Expression (RegEx) is a special sequence of characters used to search patterns inside text.
- They are a powerful tool for:
  - Checking if a specific pattern is present inside a text.
  - Parsing unstructured output from a network device.
- Online tools used for testing and learning regular expressions
  - Regexr.com (PCRE/JS engine)
  - Regex101.com (Python and many other engines)

>>> Demo

- Review Regular Expressions using Regex101.com


TextFSM
Parsing Network Device Output into Structured Data

>>> TextFSM Overview

- Python module for parsing semi-formatted text.
- Originally developed to allow programmatic access to information given by the output of CLI driven devices, such as network routers and switches
  - It can however be used for any such textual output.

>>> Using TextFSM

- The engine takes two inputs
  - Template file
  - Text input (such as command responses from the CLI of device)
- Returns a list of records that contains the data parsed from the text.
- Note: A template file is needed for each uniquely structured text input.

>>> Example: Text Input

- show vlan (Arista EOS)
- Filename: arista_eos_show_vlan.raw

```
VLAN  Name                             Status    Ports
----- -------------------------------- --------- -------------------------------
1     default                          active    Et1
10    Test1                            active    Et1, Et2
20    Test2                            suspended
30    VLAN0030                         suspended
```

>>> Example: Template File

- show vlan (Arista EOS)
- Order is important
- Filename: arista_eos_show_vlan.template

```
Value VLAN_ID (\d+)
Value NAME (\w+)
Value STATUS (active|suspended)

Start
  ^${VLAN_ID}\s+${NAME}\s+${STATUS} -> Record
```

>>> Example: Executing textfsm

```
VLAN  Name                             Status    Ports
----- -------------------------------- --------- -------------------------------
1     default                          active    Et1
10    Test1                            active    Et1, Et2
20    Test2                            suspended
30    VLAN0030                         suspended
```

```
Value VLAN_ID (\d+)
Value NAME (\w+)
Value STATUS (active|suspended)

Start
  ^${VLAN_ID}\s+${NAME}\s+${STATUS} -> Record
```

```
FSM Table:
['VLAN_ID', 'NAME', 'STATUS']
['1', 'default', 'active']
['10', 'Test1', 'active']
['20', 'Test2', 'suspended']
['30', 'VLAN0030', 'suspended']
```

>>> Using TextFSM in Python

From the previous example:

```
>>> import textfsm
>>>
>>> table = textfsm.TextFSM(open('arista_eos_show_vlan.template'))
>>>
>>> data = table.ParseText(open('arista_eos_show_vlan.raw').read())
>>>
>>>
>>> data
[['1', 'default', 'active'], ['10', 'Test1', 'active'], ['20', 'Test2', 'suspended'], ['30',
'VLAN0030', 'suspended']]
>>>
>>> table.header
['VLAN_ID', 'NAME', 'STATUS']
>>>
```

>>> Using TextFSM in Python

Parsing “show example:

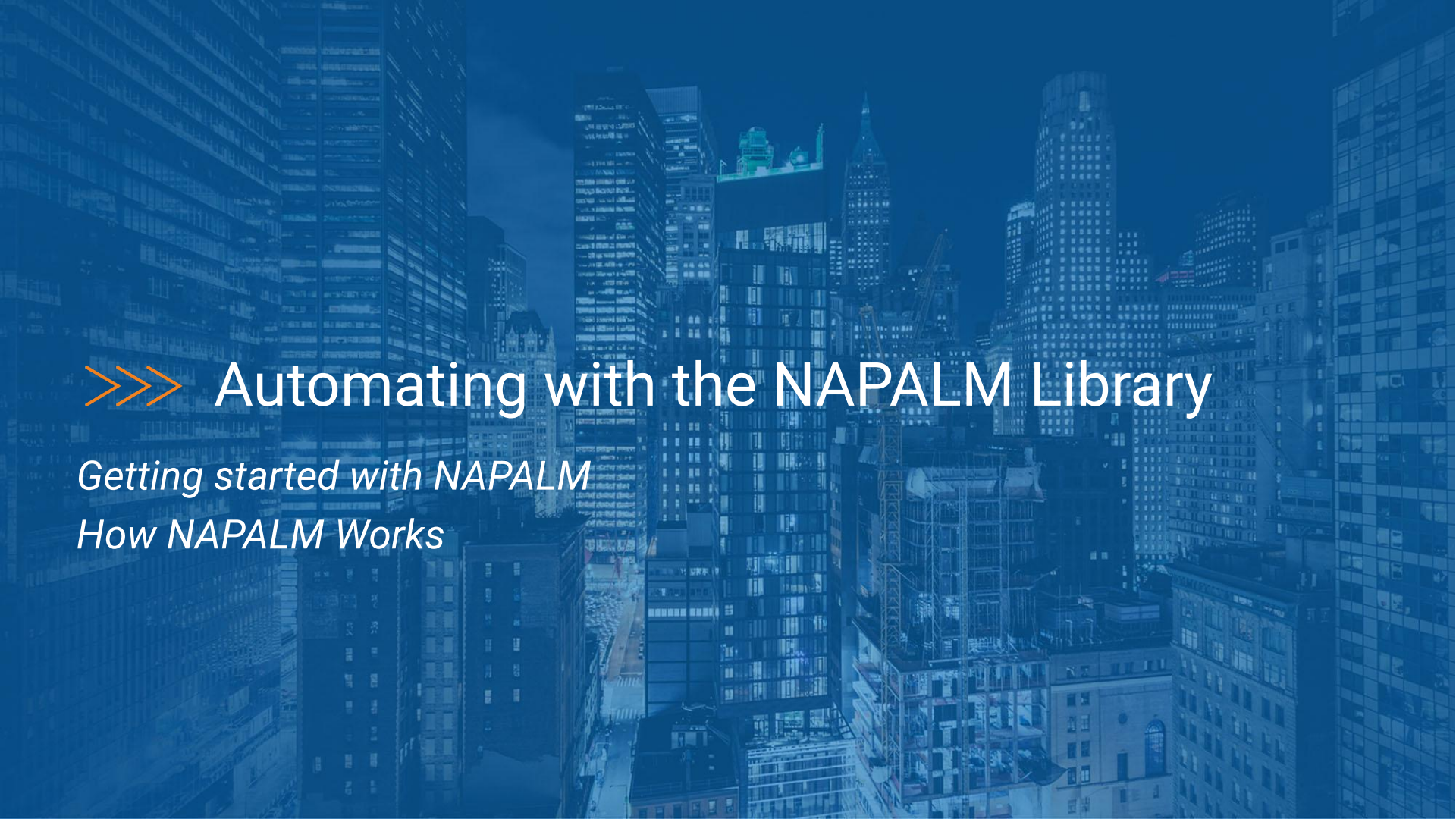
```
>>> import textfsm
>>>
>>> table = textfsm.TextFSM(open('cisco_ios_show_version.template'))
>>>
>>> data = table.ParseText(open('cisco_ios_show_version.raw').read())
>>>
>>> data
[['16.3.1', 'csr1', '2 minutes', '0x2102']]
>>>
>>> table.header
['VERSION', 'HOSTNAME', 'UPTIME', 'CONFIG_REGISTER']
>>>
```

>>> Summary

- Legacy devices are here to stay (for awhile)
- Even API-enabled device may return raw text
- Using TextFSM does not necessarily mean SSH/telnet as a transport mechanism
- Great way to bridge the gap between legacy and modern devices that return structured data

>>> Lab Time

- Bonus Lab - Parsing Show Commands with TextFSM
  - Use TextFSM to parse show ip interface brief from a Cisco Nexus switch
  - Use clitable along with netmiko to generate structured data from unstructured device output
  - Note: same workflow and process can be used for any other device.



Automating with the NAPALM Library

Getting started with NAPALM

How NAPALM Works

>>> NAPALM

- 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.
- https://napalm.readthedocs.io/en/latest/

>>> NAPALM Support Matrix

- Palo Alto PANOS
- Cisco IOS
- Cisco NX-OS
- Cisco IOS-XR
- Arista EOS
- Juniper Junos
- IBM
- Pluribus
- FortiOS
- Cumulus Linux
- Actively growing

>>> NAPALM

Three core functions:

- Retrieving Data
- Declarative Configuration Management
- Deployment Validation

All three are done in a uniform and vendor-neutral fashion

>>> Retrieving Data

Uses a uniform and consistent data model across all device types supported by NAPALM

get_facts()
get_arp_table()
get_bgp_config()
get_bgp_neighbors()
get_bgp_neighbors_detail()
get_interfaces()
get_interfaces_counters()
get_lldp_neighbors()
get_lldp_neighbors_detail()
get_ntp_peers()
get_ntp_stats()
get_ntp_servers()
… plus another dozen and growing…

>>> get_facts()

Network Device Facts

```
>>> device.get_facts()
{'os_version': u'4.15.2F-2663444.4152F', 'uptime': 5817, 'interface_list': [u'Ethernet1', u'Ethernet2',
u'Ethernet3', u'Ethernet4', u'Ethernet5', u'Ethernet6', u'Ethernet7', u'Management1'], 'vendor': u'Arista',
'serial_number': u'', 'model': u'vEOS', 'hostname': u'eos-spine1', 'fqdn': u'eos-spine1.ntc.com'}

>>> facts = device.get_facts()
>>> import json
>>> print(json.dumps(facts, indent=4))
{
    "os_version": "4.15.2F-2663444.4152F",
    "uptime": 5837,
    "interface_list": [
        "Ethernet1",
        "Ethernet2",
        "Ethernet3",
        "Ethernet4",
        "Ethernet5",
        "Ethernet6",
        "Ethernet7",
        "Management1"
    ],
    "vendor": "Arista",
    "serial_number": "",
    "model": "vEOS",
    "hostname": "eos-spine1",
    "fqdn": "eos-spine1.ntc.com"
}
```

>>> get_interfaces()

Gathering Interfaces Info

```
>>> device.get_interfaces()
{u'Management1': {'is_enabled': True, 'description': u'', 'last_flapped':
1467419703.021217, 'is_up': True, 'mac_address': u'2c:c2:60:0d:52:90', 'speed': 1000},
u'Ethernet2': {'is_enabled': True, 'description': u'', 'last_flapped': 1467419702.781202,
'is_up': True, 'mac_address': u'2c:c2:60:12:98:52', 'speed': 1000}, u'Ethernet3':
{'is_enabled': True, 'description': u'', 'last_flapped': 1467419702.781203, 'is_up': True,
'mac_address': u'2c:c2:60:60:20:9b', 'speed': 1000}, u'Ethernet1': {'is_enabled': True,
'description': u'', 'last_flapped': 1467419703.1052225, 'is_up': True, 'mac_address':
u'2c:c2:60:2d:45:d5', 'speed': 1000}, u'Ethernet6': {'is_enabled': True, 'description':
u'', 'last_flapped': 1467419702.781202, 'is_up': True, 'mac_address':
u'2c:c2:60:48:80:70', 'speed': 1000}, u'Ethernet7': {'is_enabled': True, 'description':
u'', 'last_flapped': 1467419702.8092043, 'is_up': True, 'mac_address':
u'2c:c2:60:40:8d:10', 'speed': 1000}, u'Ethernet4': {'is_enabled': True, 'description':
u'', 'last_flapped': 1467419702.769202, 'is_up': True, 'mac_address':
u'2c:c2:60:2e:c6:f8', 'speed': 1000}, u'Ethernet5': {'is_enabled': True, 'description':
u'', 'last_flapped': 1467419703.105223, 'is_up': True, 'mac_address':
u'2c:c2:60:60:7d:ba', 'speed': 1000}}
```

>>> get_interfaces() (cont'd)

```
>>> interfaces = device.get_interfaces()
>>>
>>> print(json.dumps(interfaces, indent=4))
{
    "Management1": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419703.0212176,
        "is_up": true,
        "mac_address": "2c:c2:60:0d:52:90",
        "speed": 1000
    },
    "Ethernet2": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.7812023,
        "is_up": true,
        "mac_address": "2c:c2:60:12:98:52",
        "speed": 1000
    },
    "Ethernet3": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.7812028,
        "is_up": true,
        "mac_address": "2c:c2:60:60:20:9b",
        "speed": 1000
    },
```

```
"Ethernet1": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.781203,
        "is_up": true,
        "mac_address": "2c:c2:60:48:80:70",
        "speed": 1000
    },
    "Ethernet5": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.8092043,
        "is_up": true,
        "mac_address": "2c:c2:60:40:8d:10",
        "speed": 1000
    },
    "Ethernet4": {
        "is_enabled": true,
        "description": "",
        "last_flapped": 1467419702.7692015,
        "is_up": true,
        "mac_address": "2c:c2:60:2e:c6:f8",
        "speed": 1000
    }
}
```

>>> get_interfaces_ip()

Get Interfaces IP Addresses

```
>>> {u'Management1': {u'ipv4': {u'10.0.0.11': {u'prefix_length': 24}}, u'ipv6': {}}}
{u'Management1': {u'ipv4': {u'10.0.0.11': {u'prefix_length': 24}}, u'ipv6': {}}}
>>>
```

>>> get_environment()

Device Environment Status

```
>>> device.get_environment()
{u'fans': {}, u'memory': {u'available_ram': 99060, u'used_ram': 1798476}, u'temperature': {},
u'power': {}, u'cpu': {0: {u'%usage': 5.4}}}
>>>
```

>>> NAPALM Configuration Management

Two main ways to manage device configurations with NAPALM

Configuration Replace

- Declarative configuration always pushing the full configuration
- Only commands required to get the device into its intended state are applied
- No “negation (no)” commands are sent to the device

Configuration Merge

- Send a set of commands or configuration stanza
- Only commands required to get the device into its intended state are applied
- You can use the merge for declarative management on a stanza based on OS

It does vary based on operating system.

>>> NAPALM Configuration Management

Example Workflow

Works slightly different than based on individual drivers and operating systems.

- Connect to Device
- Copy desired configuration to device (checkpoint file, candidate configuration, config session, bootflash as candidate_config.txt)
- Use a vendor command to view diffs
- Use a vendor command to apply configuration changes
- Optionally, rollback to a config that exists in the file system.

Note: you dictate if the supplied configuration is a full config file or partial configuration

>>> Configuration Replace

Focus on desired configuration commands.

Scenario: You need to remove two loopback interfaces and change the hostname.

NAPALM Config Replace: * Full configuration is sent to the device, but… * Only diffs are applied. * You do not need to worry about going from A to B - you just focus on B.

```
$ more diffs/csr1.diffs
+hostname csr1
-hostname csr_old_name
-interface Loopback100
 -ip address 1.1.1.1 255.255.255.255
-interface Loopback200
 -ip address 22.2.1.1 255.255.255.255
-ip route 10.1.1.0 255.255.255.0 192.0.1.1
```

IMPORTANT: There are zero “no” commands used. The underlying OS generates the diffs (for most NAPALM drivers).

>>> Configuration Merge

You can use NAPALM for declarative management for a sectional config too.

```
Current BGP Config
router bgp 65512
   neighbor 10.0.0.0 remote-as 65500
   neighbor 10.0.0.0 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   network 20.20.20.0/24
!
Desired BGP Config (file sent to device)
router bgp 65512
   neighbor 10.0.0.2 remote-as 65500
   neighbor 10.0.0.2 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   neighbor 10.0.0.10 remote-as 65512
   network 100.0.100.0/24
!
```

```
Diff Generated by NAPALM
    neighbor 10.0.0.0 maximum-routes 12000
    neighbor 10.0.0.1 remote-as 65512
    neighbor 10.0.0.1 maximum-routes 12000
+   neighbor 10.0.0.2 remote-as 65500
+   neighbor 10.0.0.2 maximum-routes 12000
+   neighbor 10.0.0.10 remote-as 65512
+   neighbor 10.0.0.10 maximum-routes 12000
    network 20.20.20.0/24
+   network 100.0.100.0/24
 !
 management api http-commands
    protocol http
```

>>> Configuration Merge (Advanced)

You can use NAPALM for declarative management for a sectional config too.

```
Current BGP Config
router bgp 65512
   neighbor 10.0.0.0 remote-as 65500
   neighbor 10.0.0.0 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   network 20.20.20.0/24
!
Desired BGP Config (file sent to device)
no router bgp 65512
router bgp 65512
   neighbor 10.0.0.2 remote-as 65500
   neighbor 10.0.0.2 maximum-routes 12000
   neighbor 10.0.0.1 remote-as 65512
   neighbor 10.0.0.1 maximum-routes 12000
   neighbor 10.0.0.10 remote-as 65512
   network 100.0.100.0/24
```

```
Diff Generated by NAPALM
 router bgp 65512
-   neighbor 10.0.0.0 remote-as 65500
-   neighbor 10.0.0.0 maximum-routes 12000
    neighbor 10.0.0.1 remote-as 65512
    neighbor 10.0.0.1 maximum-routes 12000
-   network 20.20.20.0/24
+   neighbor 10.0.0.2 remote-as 65500
+   neighbor 10.0.0.2 maximum-routes 12000
+   neighbor 10.0.0.10 remote-as 65512
+   neighbor 10.0.0.10 maximum-routes 12000
+   network 100.0.100.0/24
 !
Be cautious of device support. This is based on NAPALM driver
implementation which is dictated by vendor OS support. This
example is EOS.
```

>>> Getting Started with NAPALM

Step 1. Import Network Driver

```
>>> from napalm import get_network_driver
>>>
>>> driver = get_network_driver('eos')
>>>
```

Step 2. Create Device Object

```
>>> hostname = 'eos-spine1'
>>> username = 'ntc'
>>> password = 'ntc123'
>>>
>>> device = driver(hostname, username, password)
>>>
```

Step 3. Open connection

```
>>> device.open()
>>>
```

>>> Perform a Configuration Merge

Sample new config we want to send/merge with current configuration:

```
snmp.conf
snmp-server community networktocode ro
snmp-server community public ro
snmp-server community private rw
snmp-server community supersecret rw
snmp-server location SYDNEY
snmp-server contact JOHN_SMITH
```

>>> Perform a Configuration Merge

- load_merge_candidate method
- Support configuration files (filename parameter) and strings (config parameter)

```
>>> device.load_merge_candidate(filename='snmp.conf')
```

- Compare the running configuration and the new candidate configuration with compare_config

```
>>> diffs = device.compare_config()
>>>
>>> print(diffs)
@@ -7,7 +7,12 @@
 hostname eos-spine1
 ip domain-name ntc.com
 !
+snmp-server contact JOHN_SMITH
+snmp-server location SYDNEY
 snmp-server community networktocode ro
+snmp-server community private rw
+snmp-server community public ro
+snmp-server community supersecret rw
 !
 spanning-tree mode mstp
 !
```

>>> Perform a Configuration Replace

- Declarative network configuration management
- Requires using a full configuration file
- load_replace_candidate method
- Copies new config to the device, but does not commit it

>>> Other Methods

Method	Description	Example
discard_config	Removes the loaded candidate configuration	device.discard_config()
commit_config	Commit the loaded candidate configuration	device.commit_config()
rollback	Restores a backup configuration saved before the last changes and commit	device.rollback()

```
>>> device.discard_config()
>>>
>>> print(device.compare_config())
u''
>>> device.commit_config()
>>>
>>> device.rollback()
>>>
```

>>> How NAPALM Works

- EOS
  - Creates and locks config sessions
  - Uses rollback clean-config to prepare for a config replace
  - Commit is performed issuing copy startup-config flash:rollback-0, configure session # and commit
  - Rollback is performed issuing configure replace flash:rollback-0
  - Diffs are generated on the device using the show session-config named <file> diffs
- IOS
  - Uses SCP or Netmiko (TCL) to transfer config files for config replace/merge
  - Uses show archive config differences <base_file> <new_file> to show diffs for config replace
  - Uses show archive config incremental-diffs <file> ignorecase to show incremental diffs
  - Replaces with configure replace <file> force. Merges with copy <file> running-config

>>> How NAPALM Works

- Junos
  - Uses junos-pyez API with NETCONF Junos candidate configurations
  - Locks configurations while performing operations till first commit/rollback
  - Uses rollback 0 to rollback configuration
- NXOS
  - Uses checkpoint files for config replacement. A checkpoint file can be obtained with device._get_checkpoint_file() which issues checkpoint file temp_cp_file_from_napalm on the device and then prints it
  - Diffs for config replacement are a list of commands that would be needed to take the device from its current state to the desired config state using show diff rollback-patch file <source_of_truth_file> file <config_file> command
  - Merges send config line by line. This doesn't use the checkpoint/rollback functionality. As a result, merges are not atomic
  - Replaces uses rollback running file <config_file> command

>>> Lab Time - BONUS

- Bonus Lab - NAPALM
  - Using the NAPALM Python Library to do declarative config merge, full config merge and getters for Arista EOS
  - Using the NAPALM Python Library to do basic config merge and getters for Cisco IOS
  - Using the NAPALM Python Library to do declarative config merge, full config merge and getters for Juniper JUNOS