Cisco Automated Testing for your Network – pyATS v1

Created in Partnership with Solutions Readiness Engineers

Last Updated: 14-August-2020

IMPORTANT: The included documentation was not created or verified by dCloud. Check Cisco dCloud regularly for new releases!

About This Lab

This guide for the preconfigured demonstration includes:

Requirements

Required	Optional
Laptop with Cisco AnyConnect®	n/a
Software capable of connecting to Microsoft Windows Remote Desktop	n/a

Limitations

This lab doesn’t require prior experience with the pyATS library, but an intermediate level of Python knowledge helps the user to understand details of the tasks in this lab.

If you cannot finish this lab in one sitting, you can come back and start from where you left off. Tasks are written so that code in one task doesn’t require that you complete the previous task. However, we do encourage you to go through the tasks in numerical order, as later scenarios require knowledge that you gained in earlier scenarios.

Core Python knowledge that will help you understand code presented in this lab:

- Understanding object-oriented programming concepts (for example, classes, methods, objects, and inheritance)
- Basic level of understanding of decorators
- Fluent with data structures in Python (Lists and Dictionaries)
- Experience with basic concepts (logging, module imports, iterators, “for” loops, and functions)

We’ve tried to make our examples comprehensive; to provide you with a clear understanding of the pyATS concept. However, the examples are not too elaborate and are not meant for distribution in real networks.

About This Solution

During this lab, you will get hands-on experience with pyATS which is a vendor-agnostic suite of libraries for Python. You will learn how to build your automated tests, using those frameworks.

pyATS opens a wide variety of opportunities and soon you will see it's not hard to start using them.

During this lab, we will show you real-world examples that you can use to start implementing the automation of tests in your network.

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

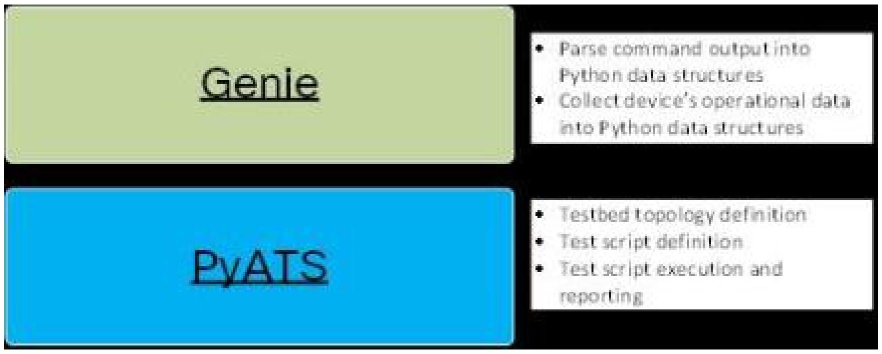
- Create a testbed file.
- Understand the main capabilities of pyATS suites.
- Start writing automated tests for your network.

Throughout this lab you will work with and learn the following features and concepts:

- pyATS testbed file
- pyATS shell
- Device.connect() and Device.execute() methods
- pyATS test script structure
- pyATS testcases
- Logging in pyATS
- pyATS Parse
- pyATS Learn
- find_links method to find all the links between devices.
- pyATS Run
- pyATS job file
- pyATS Log Viewer

About pyATS

Together, pyATS and Genie provide you with all the tools and libraries necessary for network testing automation. Let's discuss which roles pyATS and Genie play, to provide a complete ecosystem for network testing automation.


Genie
• Parse command output into Python data structures
• Collect device's operational data into Python data structures
PyATS
• Testbed topology definition
• Test script definition
• Test script execution and reporting

pyATS is the foundation of the ecosystem. It is a Python test framework, which is responsible for:

- definition of topologies and device/interconnects in a YAML file
- interaction with network devices
- definition, execution and reporting on test scripts

Genie is a pyATS library which is used to process data that is collected from networking devices by leveraging:

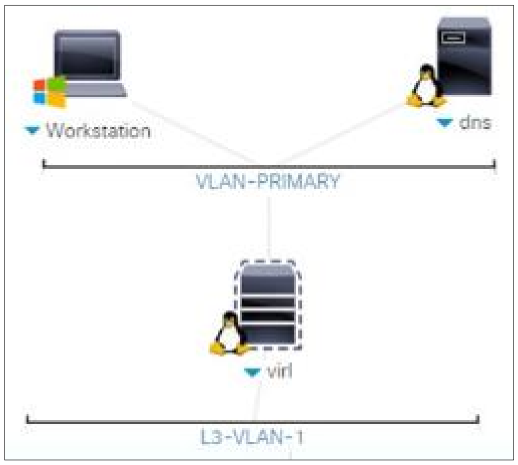
- **parsers**: converts/formats command output into Python data structures
- **learn feature**: collects protocol configuration state and operational status of a device and puts it into Python data structures.

In recent versions of pyATS, Genie became part of the pyATS package, so both pyATS and Genie go hand in hand, which makes them indistinguishable. In many cases, we use the pyATS method but under the hood, it calls Genie methods. That's why it's perfectly fine to say "pyATS" and mean "Genie." So, we will use the term "pyATS" instead of "pyATS/Genie" throughout this lab.

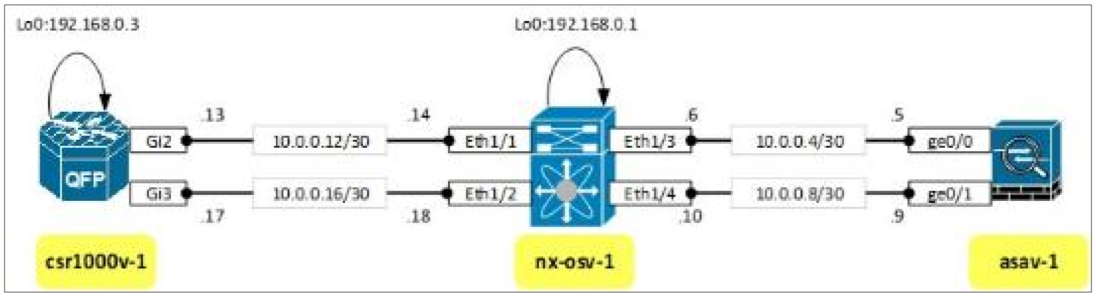
Topology

This lab includes a VIRL server on which simulation of the lab network will be running. All lab tasks will be done from the Jump host (a Windows 10 machine with WSL installed).

dCloud Topology



Topology of Simulated Networks



Equipment Details

Hostname	Credentials	IP address	Connection protocol
Jumphost	**C1sco12345**	198.18.133.252	Microsoft Remote Desktop (RDP)
asav-1	Password: **cisco** Enable Password: **cisco**	198.18.1.202	Telnet
csr1000v-1	Password: **cisco** Enable Password: **cisco**	198.18.1.201	Telnet
nx-osv-1	login: **cisco** password: **cisco**	198.18.1.203	Telnet

Component Details

Hostname	Product	Operating System
Jumphost	Microsoft Windows 10 WSL 1 with Ubuntu 18.04 Visual Studio Code	Microsoft Windows 10
asav-1	Cisco Adaptive Security Virtual Appliance	NX-OS 9.2(3)
csr1000v-1	Cisco Cloud Services Router 1000v	IOS-XE 16.9.1
nx-osv-1	Cisco Nexus 9000v Switch	ASA OS 9.9(2)

Get Started

BEFORE PRESENTING

Cisco dCloud strongly recommends that you perform the tasks in this document with an active session before presenting in front of a live audience. This will allow you to become familiar with the structure of the document and content.

It may be necessary to schedule a new session after following this guide in order to reset the environment to its original configuration.

PREPARATION IS KEY TO A SUCCESSFUL PRESENTATION.

FOR DCLOUD SCHEDULED DEMO

Follow the steps to schedule a session of the content and configure your presentation environment.

1. Initiate your dCloud session. [Show Me How]

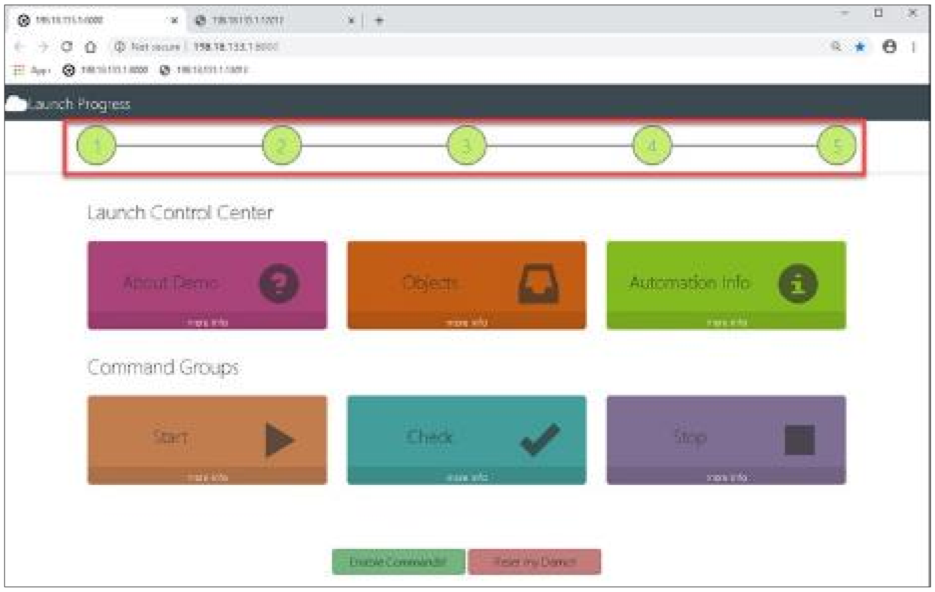
NOTE: It may take up to 10 minutes for your session to become active. After your session becomes active, **wait an additional 20 minutes** to allow for all devices to boot up properly.

2. For best performance, connect to the workstation with **Cisco AnyConnect VPN** [Show Me How] and the **local RDP client on your laptop** [Show Me How]

- Workstation 1: **198.18.133.252**, Username: **administrator**, Password: **C1sco12345**

IMPORTANT! This demonstration/lab is designed to be completed in one sitting without interruption, otherwise you may see some errors and may have to log back into the application and/or devices.

3. Wait for all stages of the Launch Progress bar to turn green before proceeding.



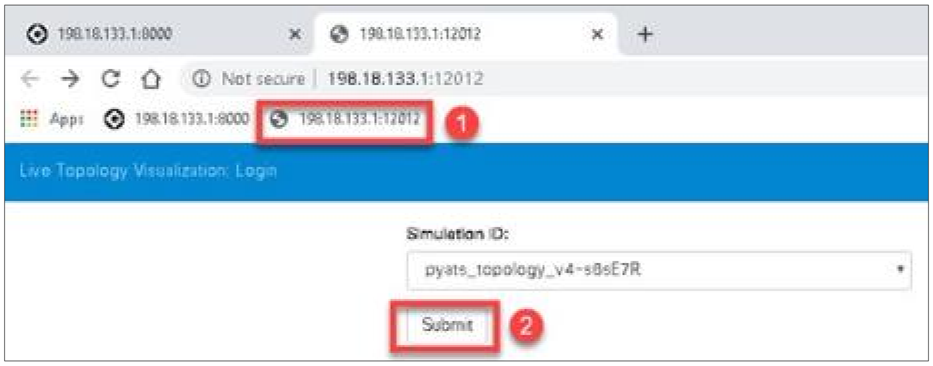
4. On the remote desktop, double-click the **PuTTY** shortcut icon.
5. Verify connectivity by launching the **asav-1**, **csr100v-1**, and **nx-osv-1** devices from the remote desktop and logging in. Username/password for all three devices: **cisco/cisco**.

- If all devices are reachable and you can log in, close the **PuTTY** sessions and proceed with **Scenario 1**.
- If only the **nx-osv-1** device is unreachable, go to the **Resolving Connectivity Issues** section.

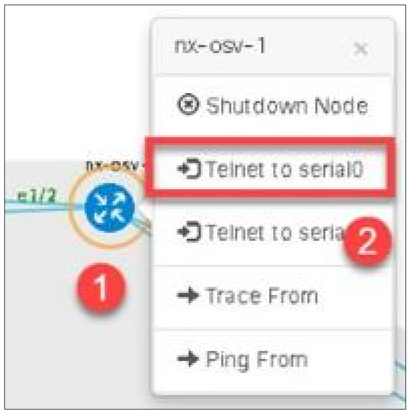
Resolving Connectivity Issues

If only the **nx-osv-1** device is unreachable, use the following procedure to resolve the connection issue.

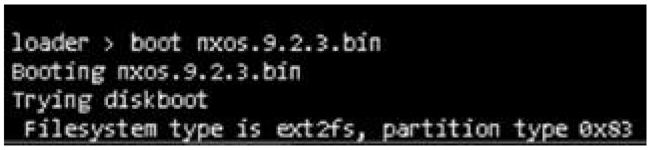
1. In Chrome, click the Live Topology Visualization: Login tab (198.18.133.1:12012).
2. Click **Submit**.



3. Click the **nx-osv-1** icon. Then select **Telnet to serial0**.



4. In the terminal window, make sure the **loader >** prompt appears. If it is present, enter **boot nxos.9.2.3.bin** to boot the NX-OS image manually.
5. Wait 10 minutes until NX-OS boots up and then try to connect to via the PuTTY shortcut on the remote desktop.



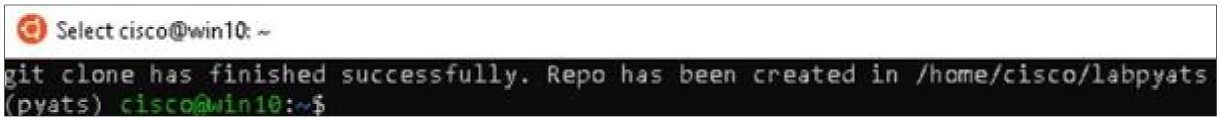
```
loader > boot nxos.9.2.3.bin
Booting nxos.9.2.3.bin
Trying diskboot
 Filesystem type is ext2fs, partition type 0x83
```

Scenario 1. Exploring the Lab Structure

Value Proposition: This scenario has you explore the lab structure to familiarize yourself with the user interface.

Steps

1. On the remote desktop, double-click the **Ubuntu 18.04 LTS** shortcut. Ubuntu will run on our RDP Jumphost on top of Windows 10. The bash Linux shell appears.



```
Select cisco@win10: ~
git clone has finished successfully. Repo has been created in /home/cisco/labpyats
(pyats) cisco@win10:~$
```

Throughout the lab, you will be working from a virtual environment. The virtual environment provides the following major advantages over running Python scripts globally:

- **Project Isolation**: Avoids installing Python packages globally which could break system tools or other projects.
- **Dependency Management**: Makes the project self-contained and reproducible by capturing all package dependencies in a requirements file.

Cisco recommends that you run pyATS scripts from the virtual environment. The keyword (pyats) at the beginning of each line indicates that you are working from a virtual environment.

2. Change to the directory that contains the lab files:

```
(pyats) cisco@win10$ cd ~/labpyats
(pyats) cisco@win10$~/labpyats
```

3. Check the lab's structure (before running the command shown below, ensure that you have changed to the correct directory: **~/labpyats**).

```
(pyats) cisco@win10:~/labpyats$ ls -l
```

Command Prompt Conventions

Throughout this lab we use the following prompt conventions at the beginning of the first line of a command or code statement:

- **$ -** commands appearing after the $ sign are input in the bash shell (be sure you copy only the commands, and not the **$** sign).
- **In [1]: -** code going after this statement is input into the pyATS shell (be sure you copy only the code, and not the **In [1]:** statement). This statement goes only on the first line of code**.**
- **Out [1]:** - signifies output in pyATS shell (in reaction to the code input into pyATS shell, which follows and **In [1]:** statement)

Doing the Lab in Visual Studio Code IDE

This lab guide is written to be used as follows:

- Bash shell in Windows Subsytem for Linux (WSL) to run pyATS interactive shell and pyATS scripts.
- Vim utility to edit the script files.

If you prefer, you can follow this lab guide with **Visual Studio Code IDE**, if you have experience with it.

NOTE: Running the lab in Visual Studio Code is recommended only if you have prior experience with Visual Studio Code IDE.

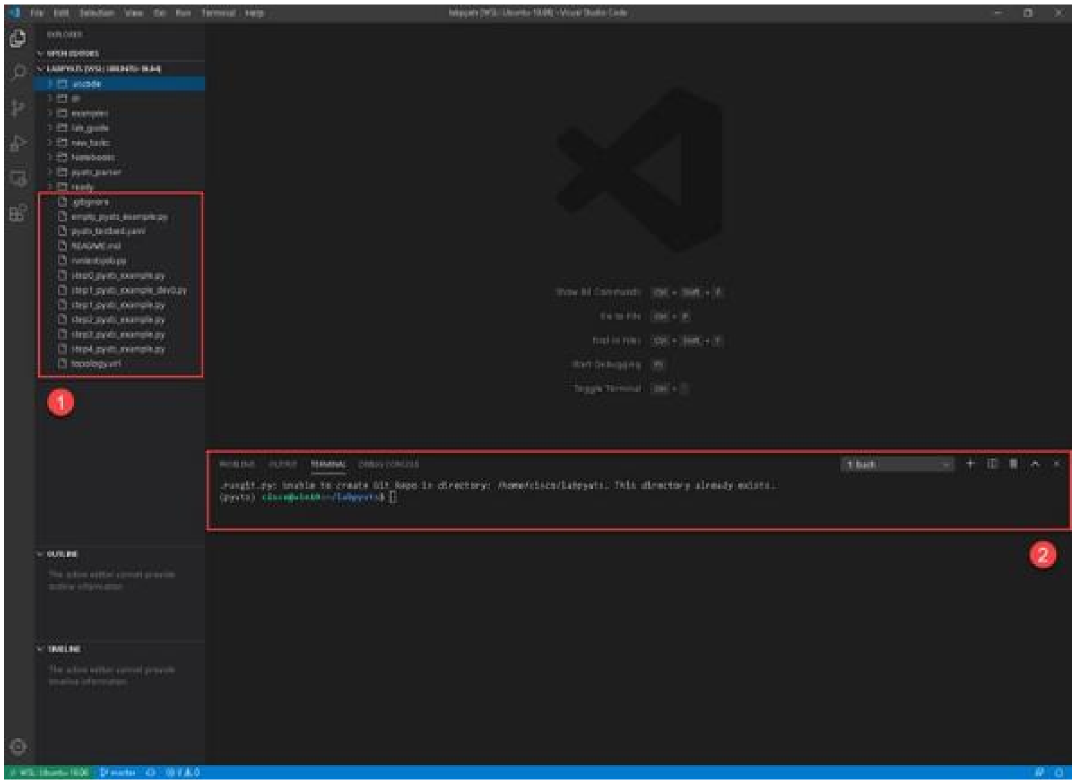
1. Enter the following command to start Visual Studio Code from Bash shell while you are in the **~/labpyats** directory.

```
$ code .
```

2. This will open a Visual Studio Code Window. Note the following:

- Observe the list of files with scripts used in this lab (section 1 of the following illustration). You can click them to open the files in Visual Studio Code editor (you can use this method instead of the Vim utility to edit the script files).
- View the terminal window that you can use to run the pyATS interactive shell and pyATS scripts, (section 2 of the following illustration), instead of a Bash shell in WSL. You will see the following message in the terminal window that is expected and can be ignored:

```
.rungit.py: Unable to create Git Repo in directory: /home/cisco/labpyats. This directory
already exists.
```



```
.rungit.py: Unable to create Git Repo in directory: /home/cisco/labpyats. This directory already exists.
(pyats) cisco@ubuntu:~/labpyats$
```

3. If you want to continue the lab in Visual Studio Code, you can close WSL at this point (type **exit** in the WSL terminal window).

NOTE: The following files are in the **labpyats** directory and will be used throughout the lab.

Filename	Description	Task #
pyats_testbed.yaml	Testbed file for pyATS (in YAML)	-
task10_labpyats.py	Test to verify reachability (ping)	Scenario 10
task11_runtestsjob.py	pyATS job file to run tests from Task 10	Scenario 11
task5_labpyats.py	The task for collection of show commands	Scenario 5
task6_labpyats.py	The initial task for pyATS exploration	Scenario 6
task71_labpyats.py	Test to verify log messages (asav-1 only)	Scenario 7
task72_labpyats.py	Test to verify log messages (all devices)	Scenario 7
task8_labpyats.py	Test to verify service contracts coverage	Scenario 8
task9_labpyats.py	Test to verify routing information	Scenario 9

This concludes scenario 1.

Scenario 2. Explore pyATS Testbed File

Value Proposition: There is a pre-configured VIRL topology file for this lab (topology.virl). Information about the network topology is defined in this file. The VIRL file contains all necessary information for pyATS: management interfaces, IP addresses, and connections between network devices. To make all this information available for pyATS, this file must be converted to testbed file in YAML format.

Steps

The Testbed YAML file for pyATS has been pre-created for this lab and is named **pyats_testbed.yaml.**

1. Open the pre-created **pyats_testbed.yaml** file in the Vim editor by entering the **vim pyats_testbed.yaml** command.

```
(pyats) cisco@win10$ cd ~/labpyats
(pyats) cisco@win10$~/labpyats $ vim pyats_testbed.yaml
```

2. The output of the command should contain the following:

```
testbed:

  name: labpyats_default_wExpSL <-- name of topology simulation

  credentials: <-- credentials for CLI access (stored as environment variables)
    default: <-- username/password used by default
      username: "%ENV{PYATS_USERNAME}"
      password: "%ENV{PYATS_PASSWORD}"
    enable: <-- enable password (if required)
      password: "%ENV{PYATS_AUTH_PASS}"
    line: <-- line (VTY/console) password (if required)
      password: "%ENV{PYATS_AUTH_PASS}"
  servers:
    jumphost:
      address: 198.18.1.100
      server: jumphost
    ~mgmt-lxc:
      address: 198.18.1.188
      server: ~mgmt-lxc

devices: <-- All necessary information to connect to devices is in this block
asav-1:
    alias: asav-1
    os: asa
    type: ASAv
    platform: ASAv

    connections:

      defaults:
        class: unicon.Unicon <-- Unicon  is Python package and used by pyATS to connect to
network devices through a command-line interface.
      console:
        protocol: telnet
        ip: 198.18.1.202
        port: 23 <-- connection to a device would be done via Telnet port
<…>

topology: <-- All information about links between devices is in this block
  asav-1:
    interfaces:
      GigabitEthernet0/0:
        ipv4: 10.0.0.5/30
        link: asav-1-to-nx-osv-1
        type: ethernet
      GigabitEthernet0/1:
        ipv4: 10.0.0.9/30
        link: asav-1-to-nx-osv-1#1
        type: ethernet
      <…>
```

Now we have all the required information to start our tests with pyATS.

NOTE: Note that username and passwords to access devices are not stored in the YAML file:

```
 credentials:
   default:
     username: "%ENV{PYATS_USERNAME}"
     password: "%ENV{PYATS_PASSWORD}"
   enable:
     password: "%ENV{PYATS_AUTH_PASS}"
   line:
     password: "%ENV{PYATS_AUTH_PASS}"
```

We recommend that you store credentials separately as environmental variables.

3. Exit Vim without saving.

```
:q!
```

4. In the lab environment, variables for PYATS with credentials to access all devices are preconfigured. Check these environment variables from a Bash shell:

```
$ echo $PYATS_USERNAME $PYATS_PASSWORD $PYATS_AUTH_PASS
```

5. The output of the command should contain the following:

```
cisco cisco cisco
```

This concludes scenario 2.

Scenario 3. Observe pyATS Capabilities using the pyATS Shell

Value Proposition: We believe it's always faster to start learning about pyATS in an interactive format where you can try different pyATS functions. It allows you to run commands and see results immediately, instead of making small changes in Python code and re-running it after every minor change.

pyATS has an interactive shell for developing tests. It's run from Bash shell in the following way:

pyats shell --testbed-file <testbed_file_name>

We will call the pyATS interactive shell pyATS shell throughout this guide.

Let's begin with the pyATS shell, using it with **pyats_testbed.yaml**, which we saw in the previous scenario.

Steps

1. Enter the following command to run the pyATS shell command from the Bash shell, and specify the YAML testbed file as the parameter. After about 10 seconds, the interactive shell opens. This is where you can input the Python code.

```
$ pyats shell --testbed-file pyats_testbed.yaml
```

2. The output of the command should contain the following (version of Python might be different):

```
Welcome to pyATS Interactive Shell
==================================
Python 3.7.5 (default, Nov  7 2019, 10:50:52)
[GCC 8.3.0]
```

3. Import required module and load testbed into Python object ("testbed"):

```
In [1]: from genie.testbed import load
testbed = load('pyats_testbed.yaml')
```

4. Check the devices included in the lab's testbed.

```
In [1]: testbed.devices
```

5. The output of the command should contain the following:

```
Out[1]: TopologyDict({'asav-1': <Device asav-1 at 0x7f5342e17210>, 'csr1000v-1': <Device
csr1000v-1 at 0x7f5342deced0>, 'nx-osv-1': <Device nx-osv-1 at 0x7f5341998890>})
```

6. Create variables (Python objects) to easily call devices (**nx - 'nx-osv-1'** device, **asa -'asav-1'** device):

```
In [1]: nx = testbed.devices['nx-osv-1']
asa = testbed.devices['asav-1']
```

7. Connect to devices from the pyATS shell:

```
In [1]: nx.connect()
asa.connect()
```

8. Let's prepare ourselves for our first test and collect **show inventory** command output from the devices.

```
In [1]: nx.execute('show inventory')
asa.execute('show inventory')
```

9. Verify the collected information in the output of each command. Pay attention to the output of both execute methods returned as plain text (string type in Python):

```
nx-osv-1#
Out[6]: 'NAME: "Chassis",  DESCR: "Nexus9000 9000v Chassis"               \r\nPID: N9K-9000v
,  VID: V02 ,  SN: 9OQ8QSK7JX1          \r\n\r\nNAME: "Slot 1",  DESCR: "Nexus 9000v Ethernet
Module"           \r\nPID: N9K-9000v           ,  VID: V02 ,  SN: 9OQ8QSK7JX1
\r\n\r\nNAME: "Fan 1",  DESCR: "Nexus9000 9000v Chassis Fan Module"    \r\nPID: N9K-9000v-FAN
,  VID: V01 ,  SN: N/A                  \r\n\r\nNAME: "Fan 2",  DESCR: "Nexus9000 9000v
Chassis Fan Module"    \r\nPID: N9K-9000v-FAN       ,  VID: V01 ,  SN: N/A
\r\n\r\nNAME: "Fan 3",  DESCR: "Nexus9000 9000v Chassis Fan Module"    \r\nPID: N9K-9000v-FAN
,  VID: V01 ,  SN: N/A'
asav-1#
Out[7]: 'Name: "Chassis", DESCR: "ASAv Adaptive Security Virtual Appliance"\r\nPID: ASAv
, VID: V01     , SN: 9AWXBH2QJP7'
```

10. Exit the pyATS shell by using the **exit** command.

This concludes scenario 3.

Scenario 4. Collect Show Commands from the Network Devices

Value Proposition: In this task, we will use the knowledge we have gained from the previous task to write a script that collects basic 'show' commands from each device in the testbed. The output of these commands will be saved in the directory created by the script: **gathered_commands**. These outputs can be used later if you want to compare the future state of the network with the current one. Since it's required to collect outputs from all the devices in the testbed, in this task we will work with the **testbed.devices** object, and iterate over all the devices contained in this object to collect an output of the required commands from each device.

Steps

1. Let's connect to pyATS and check parts of the code before running the final script. In the beginning, we will check the structure of **testbed.devices** object.

```
$ pyats shell --testbed-file pyats_testbed.yaml

In [1]: print(testbed.devices)
```

2. Check the output.

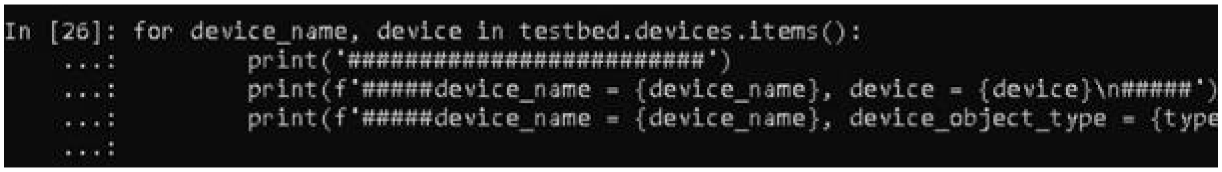
```
Out[1]: TopologyDict({'asav-1': <Device asav-1 at 0x7f60bef1da90>, 'csr1000v-1': <Device
csr1000v-1 at 0x7f60beee73d0>, 'nx-osv-1': <Device nx-osv-1 at 0x7f60bda8d850>})
```

NOTE: As you can see from the output in the previous step, **'Device <device_name>'** objects are contained as dictionary values in the object of **TopologyDict** class. The device names are used as dictionary keys.

Let's try to apply standard dictionary method: **items()** to get **keys** (device names) and **values** (respective device objects). For iteration, the **for** loop will be used:

- **device_name** variable will be used to store a device name.
- **device variable** will be used to store the respective device object.

NOTE: Before running it, note the Python indentation. An example of the output is shown below.



```
In [26]: for device_name, device in testbed.devices.items():
    ...:         print('#############################')
    ...:         print(f'#####device_name = {device_name}, device = {device}\n#####')
    ...:         print(f'#####device_name = {device_name}, device_object_type = {type
    ...:
```

3. Paste the code shown below into the pyATS console.

NOTE: Because this lab guide is in PDF format, it doesn't allow the use of copy and paste for proper indentation in the pyATS interactive shell. To copy and paste the code snippet below, use the appropriate section in the **cisco_automated_testing-11082020.txt** file provided with this lab. A straight copy and paste from the PDF will not work.

```
In [1]: for device_name, device in testbed.devices.items():
    print('#########################')
print(f'#####device_name = {device_name}, device = {device}\n#####')
print(f'#####device_name = {device_name}, device_object_type =
{type(device)}\n#########################')
```

4. Check the output to understand the structure of **TopologyDict** better. The output for each device is shown inside '#########################' stanzas. Pay special attention to the highlighted pars; pyATS knows about the device type (ASAv, CSR1000v, NX-OSv 9000) and creates an object with the different device type for it.

```
#########################
#####device_name = asav-1, device = Device asav-1, type ASAv
#####
#####device_name = asav-1, device_object_type = <class 'genie.libs.conf.device.Device'>
####
#####################
#########################
#####device_name = csr1000v-1, device = Device csr1000v-1, type CSR1000v
#####
#####device_name = csr1000v-1, device_object_type = <class
'genie.libs.conf.device.iosxe.device.Device'>
#########################
#########################
#####device_name = nx-osv-1, device = Device nx-osv-1, type NX-OSv 9000
#####
#####device_name = nx-osv-1, device_object_type = <class
'genie.libs.conf.device.nxos.device.Device'>
#########################
```

5. Since our script shows that it iterates over all the devices correctly, let's try to connect to each device and run the **show inventory** command to test the logic.

- **device.connect()** method will be used to connect to device (same as in the previous task).
- **device.execute()** method will be used (same as in the previous task).

```
In [1]: device.connect()
print(device.execute('show inventory'))
```

6. Let's combine this code with what has been run in the previous code to check the output of **device.execute('show inventory')**. The added code is highlighted. Copy and paste the code into the pyATS console.

NOTE: If a device connection is closed or terminated unexpectedly after it has already connected to a device, there will be multiple errors generated (for example, the Python **EOF** exception would be invoked) at the time of command execution.

To handle this situation, it's required to add the following code to reconnect to a device in case a broken connection to a device has been detected:

```
from unicon.core.errors import EOF, SubCommandFailure

try:

    device.execute('show inventory')

except SubCommandFailure as e:

    if isinstance(e.__cause__, EOF):

        print('Connection closed, try reconnect')

        device.disconnect()

        device.connect()
```

Also refer to the "EOF Exception handling" chapter in the following document for more details:

https://pubhub.devnetcloud.com/media/unicon/docs/user_guide/services/generic_services.html

NOTE: Because this lab guide is in PDF format, it doesn't allow the use of copy and paste for proper indentation in the pyATS interactive shell. To copy and paste the code snippet below, use the appropriate section in the **cisco_automated_testing-11082020.txt** file provided with this lab. A straight copy and paste from the PDF will not work.

To paste the code from the **cisco_automated_testing-11082020.txt**, saving the indentation, follow the procedure (see the illustration below for an example):

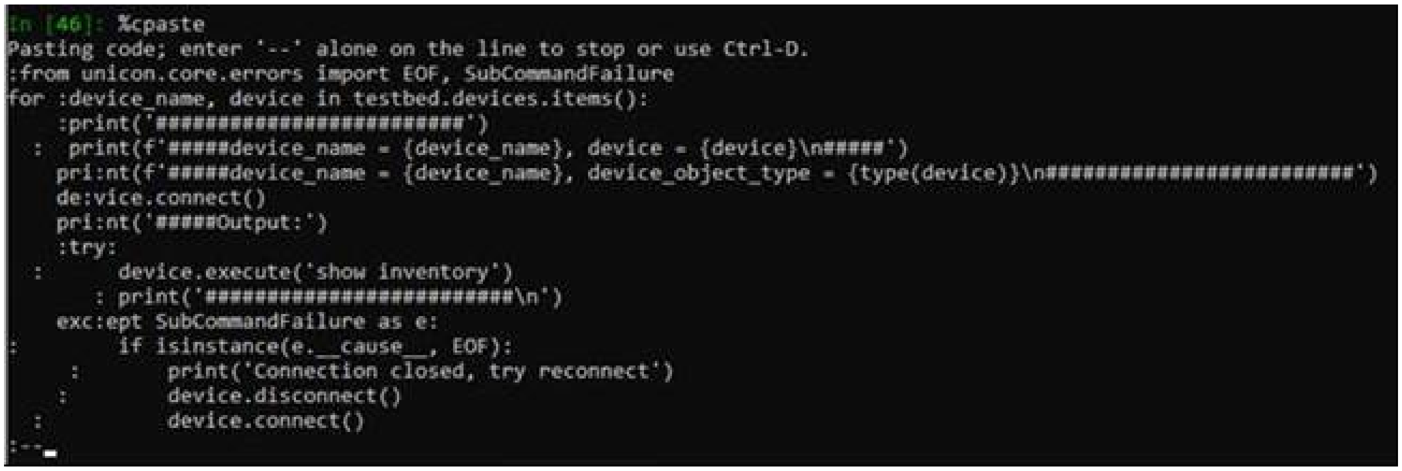
a. Place the following iPython command in the beginning of code:

```
%cpaste
```

b. <paste the code>

c. End the code with --

```
In [1]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.

from unicon.core.errors import EOF, SubCommandFailure
for device_name, device in testbed.devices.items():
    print('#########################')
    print(f'#####device_name = {device_name}, device = {device}\n#####')
    print(f'#####device_name = {device_name}, device_object_type = 
{type(device)}\n#########################')
    device.connect()
    print('#####Output:')
    try:
        device.execute('show inventory')
        print('########################\n')
    except SubCommandFailure as e:
        if isinstance(e.__cause__, EOF):
            print('Connection closed, try reconnect')
            device.disconnect()
            device.connect()
--
```



```
In [46]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:from unicon.core.errors import EOF, SubCommandFailure
for :device_name, device in testbed.devices.items():
    :print('##########################')
  :  print(f'#####device_name = {device_name}, device = {device}\n#####')
    pri:nt(f'#####device_name = {device_name}, device_object_type = {type(device)}\n##########################')
    de:vice.connect()
    pri:nt('#####Output:')
    :try:
  :      device.execute('show inventory')
       : print('##########################\n')
    exc:ept SubCommandFailure as e:
:        if isinstance(e.__cause__, EOF):
     :       print('Connection closed, try reconnect')
    :        device.disconnect()
  :          device.connect()
:--
```

7. Check the result of this code. Now each device should return the output of **'show inventory'** command. Pay special attention to the message stating there is a connection to the device already. This is because we have connected to it previously. This connection is left open until the pyATS shell session is finished.

```
########################
#####device_name = asav-1, device = Device asav-1, type ASAv
#####
#####device_name = asav-1, device_object_type = <class 'genie.libs.conf.device.Device'>
########################
#####Output:
[2019-10-23 06:17:28,939] asav-1 is already connected
[2019-10-23 06:17:28,941] +++ asav-1: executing command 'show inventory' +++
Warning: ASAv platform license state is Unlicensed.
Install ASAv platform license for full functionality.
show inventory
Name: "Chassis", DESCR: "ASAv Adaptive Security Virtual Appliance"
PID: ASAv                , VID: V01     , SN: 9AP535X8NA1
asav-1#
########################
<..>
```

8. For the script collecting many commands, it would be preferred to prune the output of the commands to the console. We will use this approach with the final version of the script in this step and all scripts in the later tasks.

```
In [1]: device.connect(log_stdout=False)
```

NOTE:

log_stdout=False will disable all logging to a screen for the whole connection session to this device (until disconnect takes place) or until log_stdout is set to **True.**

log_stdout works during the initialization of a connection.

After the connection established, it's required to use device.log_user(enable=True or False) to control the logging behavior.

9. Now let's prepare a simple Python dictionary with the following structure.

- **key**: the operating system of a device (corresponds to the value in **'device.os'** field).
- **value**: Python list of string. Each string contains commands to run on the device.

```
In [1]: commands_to_gather = {
'asa': ['show inventory', 'show route'],
'iosxe': ['show inventory', 'show ip route vrf *'],
'nxos': ['show inventory', 'show ip route vrf all']
}
```

10. After the connection to each device has been established, the script will check if this device type is specified in the **commands_to_gather** dictionary. If so, it will collect all the commands for this device type. All the other additions to the code from the previous step are highlighted. Copy and paste this code to the pyATS shell. Ensure the result contains the output of both commands from the list for each device type.

NOTE: The same issue mentioned in step 6 previously may occur here in step 10, as well:

If a device connection is closed or terminated unexpectedly after connecting to the device, there will be multiple errors generated (for example, the Python **EOF** exception would be invoked) at the time of command execution.

To handle this situation, the same try/except Python construction is used to reconnect to a device in case a broken connection to a device has been detected.

NOTE: Because this lab guide is in PDF format, it doesn't allow the use of copy and paste for proper indentation in the pyATS interactive shell. To copy and paste the code snippet below, use the appropriate section in the **cisco_automated_testing-11082020.txt** file provided with this lab. A straight copy and paste from the PDF will not work.

To paste the code from the **cisco_automated_testing-11082020.txt**, saving the indentation, use the following procedure. Also refer to the illustration that follows for an example.

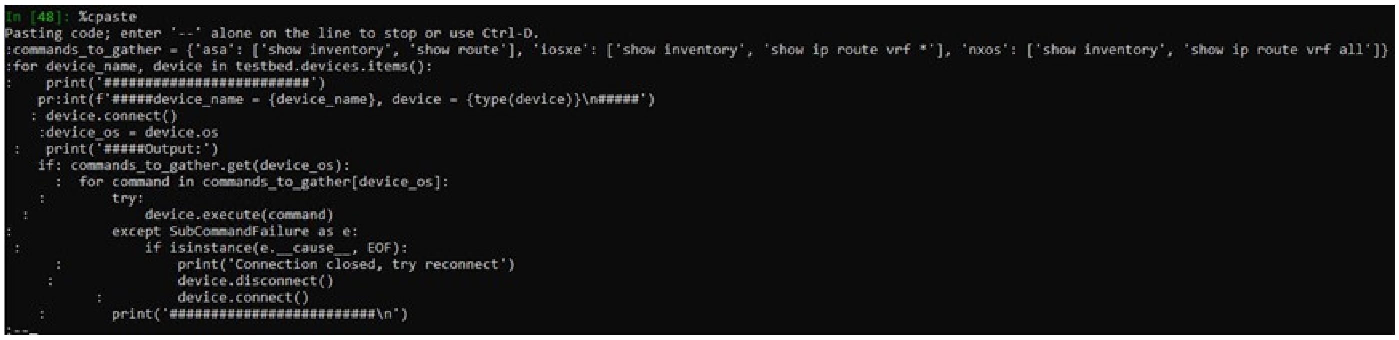
a. Place the following iPython command in the beginning of code:

```
%cpaste
```

b. <paste the code>

c. End the code with **--**

```
In [1]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
commands_to_gather = {'asa': ['show inventory', 'show route'], 'iosxe': ['show inventory',
'show ip route vrf *'], 'nxos': ['show inventory', 'show ip route vrf all']}
for device_name, device in testbed.devices.items():
    print('########################')
    print(f'#####device_name = {device_name}, device = {type(device)}\n#####')
    device.connect()
    device_os = device.os
    print('#####Output:')
    if commands_to_gather.get(device_os):
        for command in commands_to_gather[device_os]:
            try:
                device.execute(command)
            except SubCommandFailure as e:
                if isinstance(e.__cause__, EOF):
                    print('Connection closed, try reconnect')
                    device.disconnect()
                    device.connect()
            print('########################\n')
--
```



```
In [48]: %cpaste
Pasting code; enter '--' alone on the line to stop or use Ctrl-D.
:commands_to_gather = {'asa': ['show inventory', 'show route'], 'iosxe': ['show inventory', 'show ip route vrf *'], 'nxos': ['show inventory', 'show ip route vrf all']}
:for device_name, device in testbed.devices.items():
:    print('########################')
    pr:int(f'#####device_name = {device_name}, device = {type(device)}\n#####')
   : device.connect()
    :device_os = device.os
 :   print('#####Output:')
    if: commands_to_gather.get(device_os):
      :  for command in commands_to_gather[device_os]:
     :       try:
  :              device.execute(command)
:            except SubCommandFailure as e:
 :               if isinstance(e.__cause__, EOF):
      :                  print('Connection closed, try reconnect')
     :                   device.disconnect()
           :             device.connect()
     :       print('########################\n')
:--
```

11. Exit the pyATS shell by using the **exit** command. Now we are ready to go through the final version of the script gathering commands specified from all the devices in the testbed and saving them to files on Linux.

NOTE: The main logic of the script is based on the previous steps.

12. Open the prepared script **task5_labpyats.py** in Vim editor.

```
$ vim task5_labpyats.py
```

13. Before diving into the details of the code, please, study the explanation of the code given below.

The script **task5_labpyats.py** has the following Python functions:

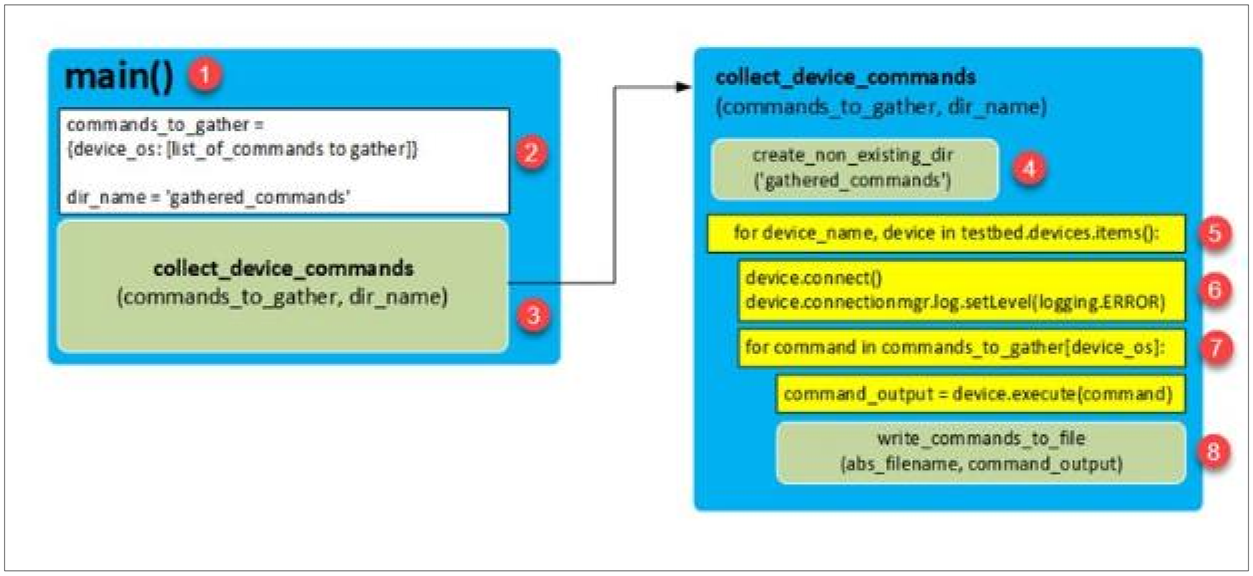
- **def main()** – see "1" on the illustration that follows. This function:
- Contains the **commands_to_gather** dictionary, where the list of commands for each device type is stored (see "2").
- Calls the **collect_device_commands** function (see "3").
- **def create_non_existing_dir(dir_path)** – see "4". This function is supplementary and is used to check whether the directory already exists. If it does not, the function tries to create it. In the event that it can't be created, the script will throw an error and exit.
- **def collect_device_commands(commands_to_gather, dir_name)** – the function that does most of the job. The main tasks this function performs include the following:
- Creates a directory for the gathered commands (calls create_non_existing_dir function for it – see "4")
- Iterates over devices - see "5" (in the way it has been done in the previous steps). For each device:

- Connects to the device (see “6”).
- Iterates over the list of commands for the respective device type. Collects all the commands specified for the device of this type (list of commands for each device type is taken from **commands_to_gather** dictionary) – see “7”.
- Writes output of the commands for this device to a file (output of each command goes to a distinct file) – see “8”.
- **def write_commands_to_file(abs_filename, command_output)** – this is a supplementary function and it’s used to write the output of commands to a file (see “8”).

NOTE: To simplify the script, the name of the testbed is hard-coded into the main():

testbed_filename = './pyats_testbed.yaml'

In further scripts, the name of a testbed will be input as a parameter of the script.



```
commands_to_gather =
{device_os: [list_of_commands to gather]}

dir_name = 'gathered_commands'

collect_device_commands
(commands_to_gather, dir_name)
```

```
create_non_existing_dir
('gathered_commands')
for device_name, device in testbed.devices.items():
    device.connect()
    device.connectionmgr.log.setLevel(logging.ERROR)
    for command in commands_to_gather[device_os]:
        command_output = device.execute(command)
        write_commands_to_file
        (abs_filename, command_output)
```

14. Now run the script:

```
$ python3 task5_labpyats.py
```

15. Check that there is a new directory created: **gathered_commands**. Check the time when it was created.

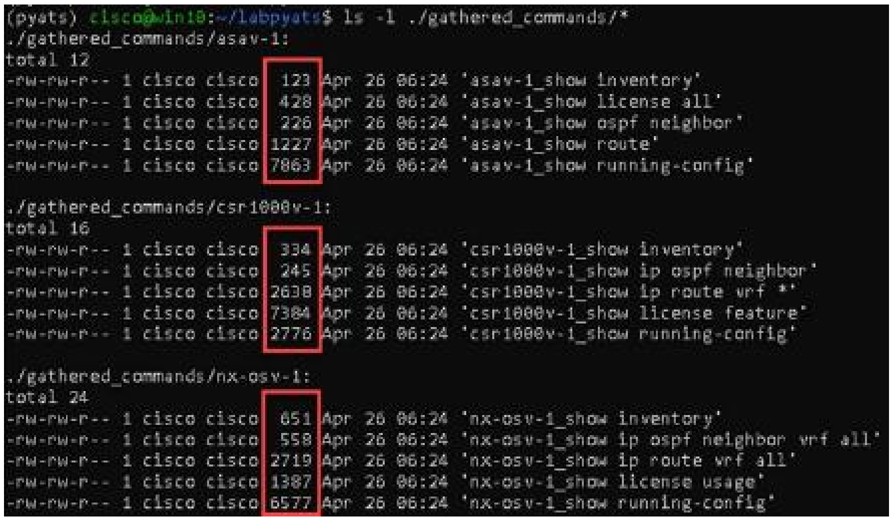
```
$ ls -l ~/labpyats | grep gathered_commands
```

Sample output in Bash shell:

```
drwxrwxrwx 1 cisco cisco    512 Jan 23 12:08 gathered_commands
```

16. Check all the files contained in the sub-directories of the **gathered_commands** directory. Ensure each file has a name in the format**: <device_name>_<command>** and a size greater than 0.

```
$ ls -l ~/labpyats/gathered_commands/*
```



```
(pyats) cisco@win10:~/labpyats$ ls -l ./gathered_commands/*
./gathered_commands/asav-1:
total 12
-rw-rw-r-- 1 cisco cisco  123 Apr 26 06:24 'asav-1_show inventory'
-rw-rw-r-- 1 cisco cisco  428 Apr 26 06:24 'asav-1_show license all'
-rw-rw-r-- 1 cisco cisco  226 Apr 26 06:24 'asav-1_show ospf neighbor'
-rw-rw-r-- 1 cisco cisco 1227 Apr 26 06:24 'asav-1_show route'
-rw-rw-r-- 1 cisco cisco 7863 Apr 26 06:24 'asav-1_show running-config'

./gathered_commands/csr1000v-1:
total 16
-rw-rw-r-- 1 cisco cisco  334 Apr 26 06:24 'csr1000v-1_show inventory'
-rw-rw-r-- 1 cisco cisco  245 Apr 26 06:24 'csr1000v-1_show ip ospf neighbor'
-rw-rw-r-- 1 cisco cisco 2638 Apr 26 06:24 'csr1000v-1_show ip route vrf *'
-rw-rw-r-- 1 cisco cisco 7384 Apr 26 06:24 'csr1000v-1_show license feature'
-rw-rw-r-- 1 cisco cisco 2776 Apr 26 06:24 'csr1000v-1_show running-config'

./gathered_commands/nx-osv-1:
total 24
-rw-rw-r-- 1 cisco cisco  651 Apr 26 06:24 'nx-osv-1_show inventory'
-rw-rw-r-- 1 cisco cisco  558 Apr 26 06:24 'nx-osv-1_show ip ospf neighbor vrf all'
-rw-rw-r-- 1 cisco cisco 2719 Apr 26 06:24 'nx-osv-1_show ip route vrf all'
-rw-rw-r-- 1 cisco cisco 1387 Apr 26 06:24 'nx-osv-1_show license usage'
-rw-rw-r-- 1 cisco cisco 6577 Apr 26 06:24 'nx-osv-1_show running-config'
```

This concludes scenario 4.

Scenario 5. Write the Test Script using pyATS Library

Value Proposition: Now let's write our first test script on Python using the pyATS library. Our test script will connect to all the devices in the testbed and print the results of the connection. If the connections to all the devices are successful, then the test will pass, else it will fail. Using this simple test script, we will learn the structure of the pyATS test script file.

The pyATS test script is a file with the Python code which uses the pyATS library.

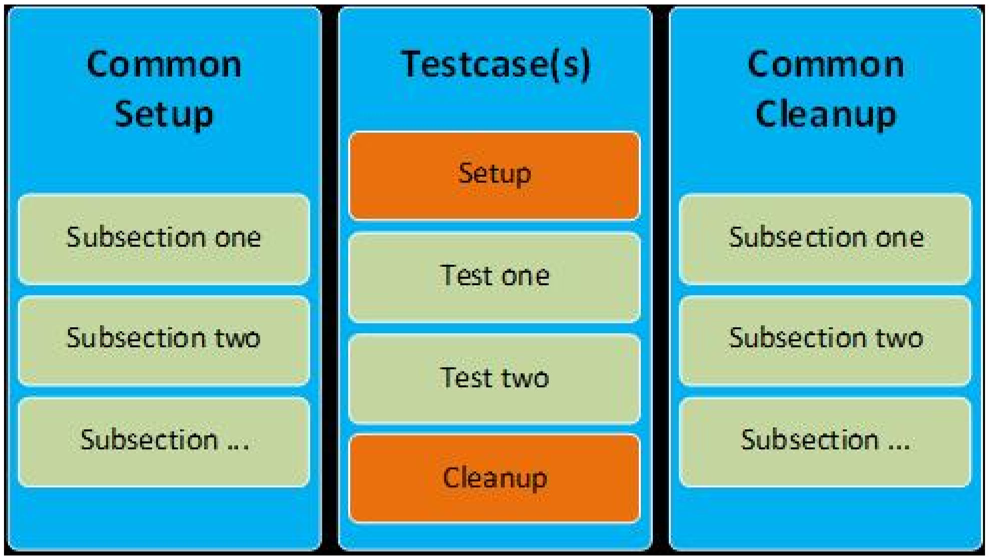
The structure of the pyATS test script is modular and straightforward.

Each test script is written in a Python file and split into three major sections (Python classes) – see the illustration below for a graphical representation:

- Common Setup: The first section in the test script, run at the beginning. It performs all the "common" setups required for the script.
- Testcase(s): A self-contained individual unit of testing. Each testcase is independent of the other testcases.
- Common Cleanup: The last section in the test script, performs all the "common" cleanups at the end of execution.

Each of these sections is further broken down into smaller subsections (Python methods of the class).

NOTE: Both Common Setup and Common Cleanup could be only one in a script, whereas there might be multiple test cases in one test script.



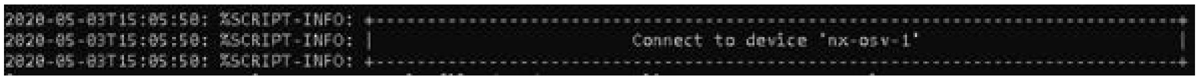
1. Let's verify our first test script. Open the file **task6_labpyats.py** and observe its structure:

   ```
   $ vim task6_labpyats.py
   ```

2. Pay special attention to the following part of the code. Whereas it's not related to only this task, it will help you understand the logging capabilities of pyATS that would be used in other tasks in this lab:

   ```
   # Import of pyATS logging banner
   from pyats.log.utils import banner

   <..>

   # This section sets up logging (ensure the log.level is the same or higher than
   # level of log.info where banner is used)

   global log
   log = logging.getLogger(__name__)
   log.setLevel(logging.INFO)
   <..>

   # Use pyATS logging banner to format the output
   log.info(banner(f"Connect to device '{device.name}'"))
   ```

3. When the pyATS logging banner is used, the following format of message would be shown in the output of the test.



```
2020-05-03T15:05:50: %SCRIPT-INFO: +------------------------------------------------------------------------------+
2020-05-03T15:05:50: %SCRIPT-INFO: |                            Connect to device 'nx-osv-1'                      |
2020-05-03T15:05:50: %SCRIPT-INFO: +------------------------------------------------------------------------------+
```

NOTE: The pyATS logging banner itself does not perform logging, and instead only performs style formatting of its input messages. Hence, the **log.info(banner("logging message"))** construction is used in the code for logging.

Since the banner is logged with INFO logging level, it's required to set logging level up to INFO (default is WARNING):

log.setLevel(logging.INFO)

4. Let's look at the main contents of this example.

   ```
   # Import of pyATS library
   from pyats import aetest

   # Genie Imports
   from genie.conf import Genie
   ```

Python class **common_setup** which is inherited from **aetest.CommonSetup** represents the major section "Common Setup" (see the following illustration). The Python class **common_setup** is where initializations and preparations before the actual script's testcases should be performed. For this reason, code in class **common_setup** is always run first, before all the testcases.

Refer to the description of the code of this Python class shown below:

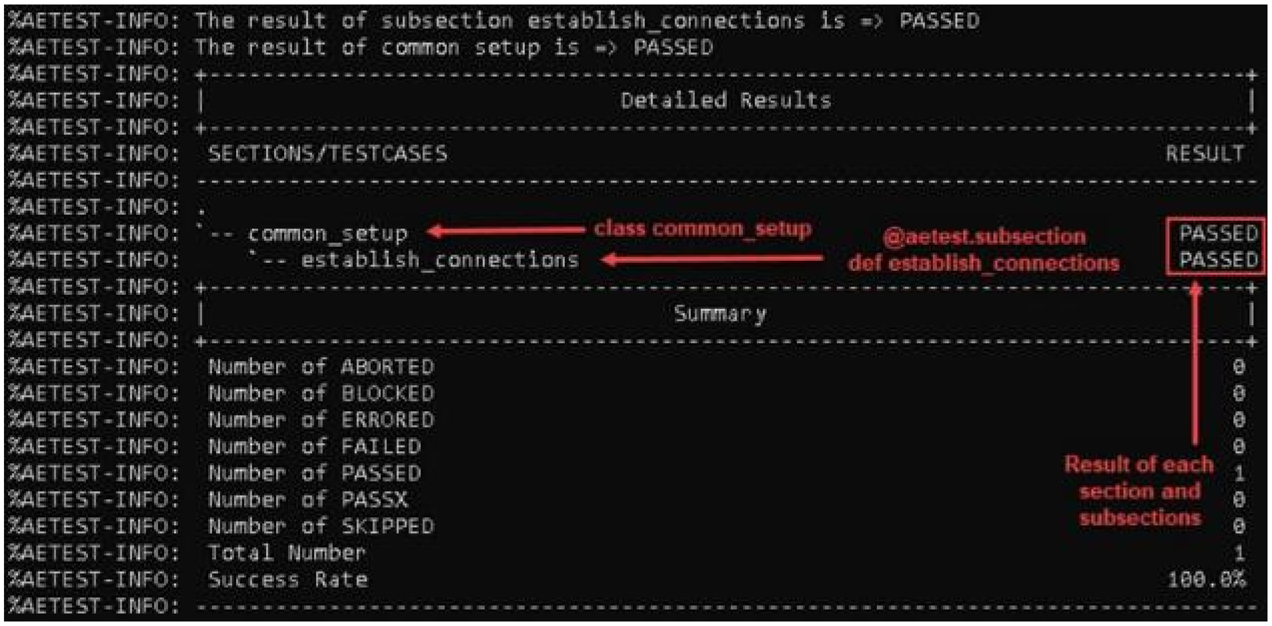
class common_setup(aetest.CommonSetup):

```
@aetest.subsection
    def establish_connections(self, testbed):
        # Load testbed file which is passed as command-line argument
        genie_testbed = Genie.init(testbed)
        self.parent.parameters['testbed'] = genie_testbed
        device_list = []
        # Load all devices from testbed file and try to connect to them
        for device in genie_testbed.devices.values():
            log.info(banner(f"Connect to device '{device.name}'"))
            try:
                device.connect(log_stdout=False)
            except errors.ConnectionError:
                self.failed(f"Failed to establish connection to '{device.name}'")
            device_list.append(device)
        # Pass list of devices to testcases
        self.parent.parameters.update(dev=device_list)
```

5. Let's run our first test script. This test script will try to connect to all the devices in the testbed and print the results of these attempts:

```
$ python3 task6_labpyats.py --testbed pyats_testbed.yaml
```

6. Upon finishing the test script, pyATS generates a report of Success/Failed testcases, the **common_setup** section is also treated as the testcase with subsection **establish_connections**. Since all the devices are reachable, the testcases should finish successfully (PASSED). Refer to the following illustration.



```
%AETEST-INFO: The result of subsection establish_connections is => PASSED
%AETEST-INFO: The result of common setup is => PASSED
%AETEST-INFO: +------------------------------------------------------------------------------+
%AETEST-INFO: |                            Detailed Results                                  |
%AETEST-INFO: +------------------------------------------------------------------------------+
%AETEST-INFO:  SECTIONS/TESTCASES                                                      RESULT
%AETEST-INFO: --------------------------------------------------------------------------------
%AETEST-INFO: .
%AETEST-INFO: `-- common_setup                                                          PASSED
%AETEST-INFO:     `-- establish_connections                                             PASSED
%AETEST-INFO: +------------------------------------------------------------------------------+
%AETEST-INFO: |                               Summary                                        |
%AETEST-INFO: +------------------------------------------------------------------------------+
%AETEST-INFO:  Number of ABORTED                                                            0
%AETEST-INFO:  Number of BLOCKED                                                            0
%AETEST-INFO:  Number of ERRORED                                                            0
%AETEST-INFO:  Number of FAILED                                                             0
%AETEST-INFO:  Number of PASSED                                                             1
%AETEST-INFO:  Number of PASSX                                                              0
%AETEST-INFO:  Number of SKIPPED                                                            0
%AETEST-INFO:  Total Number                                                                 1
%AETEST-INFO:  Success Rate                                                            100.0%
%AETEST-INFO: --------------------------------------------------------------------------------
```

class common_setup
@aetest.subsection
def establish_connections
Result of each section and subsections

This concludes scenario 5.

Scenario 6. Verify Log Messages

Value Proposition: In this test case, we will verify that the logging messages with ERROR or WARN are not present on the devices in the testbed.

The high-level logic of the test case will be as follows:

- Connect to each device in the testbed.
- Collect the output of **show logging | i ERROR|WARN**.
- If the output contains more than 0 strings, then ERROR messages were found and the test should fail for this device. Otherwise, the test should succeed.

1. Before creating our testcase, connect to ASA. Launch **PuTTY** and connect to **asav-1**.

```
User Access Verification

Password: cisco
asav-1> enable
Password: cisco
asav-1#
asav-1# clear logging buffer
asav-1#
```

2. Let's open the pyATS shell and check it out.

```
$ pyats shell --testbed-file pyats_testbed.yaml
```

3. Input the following code into pyATS shell:

```
In [1]: csr = testbed.devices['csr1000v-1']
asa = testbed.devices['asav-1']
csr.connect()
asa.connect()
```

4. Let's verify whether there are any errors or warning messages in the logs:

```
In [1]: out1 = csr.execute('show logging | i ERROR|WARN')
out2 = asa.execute('show logging | i ERROR|WARN')
```

NOTE: The output for ASA should be empty.

If you don't see any ERROR logs on the CSR1000v-1 device, then:

1. Connect to CSR:

Launch PuTTY and connect to **csr1000v-1**. Username: **cisco**, password: **cisco**

2. Generate a test ERROR message:

csr1000v-1# **send log 'Test ERROR message for pyATS'**

3. Repeat step 3 above for CSR in the pyATS shell:

out1 = csr.execute('show logging | i ERROR|WARN')

5. To check whether there is an empty or non-empty output, we will use the Python **len()** built-in function, which returns the length of the given string. If the collected output is empty, then **len()** of the output will be 0, otherwise, the result will be greater than 0.

- Input into pyATS shell:

```
In [1]: len(out2)
```

- The resulting length is 0, means output from ASA is empty:

```
Out [1]: 0
```

- Input into pyATS shell:

```
In [1]: len(out1)
```

- The resulting length is greater than 0, which means the output from CSR is not empty:

```
Out [1]: 664
```

6. Exit pyATS shell using the **exit** command.
7. Open the file **task71_labpyats.py** in Vim editor:

```
$ vim task71_labpyats.py
```

This file reuses the **establish_connections(self, testbed)** method from **task6_labpyats.py** (used in previous scenarios), which make connections to all devices in the testbed.

NOTE: Pay special attention to the method **self.parent.parameters.update(dev=device_list)**, located at the end of the **establish_connections(self,testbed)** method. Where **self.parent.parameters** is an attribute of class **aetest**, and **aetest** is the class all the testcase classes and **MyCommonSetup** class are inherited from:

class MyCommonSetup(aetest.CommonSetup):

<...>

class VerifyLogging(aetest.Testcase):

<...>

Using **self.parent.parameters** attribute arguments can be passed between different classes.

As an example, in the class **MyCommonSetup**, we store all the devices from the variable **device_list** in the parameter **parameters['dev']**.

self.parent.parameters.update(dev=device_list)

Then we can access all the devices in class VerifyLogging, using the method **self.parent.parameters['dev']**.

8. Pay special attention to the code in class **VerifyLogging**, which is used to implement the approach that has been tested using the pyATS shell.

NOTE: device.connect(log_stdout=False) is used in this example (see **def establish_connections**).

This code **(log_stdout=False) -** disables all logging to a screen for the whole connection session. To make the execution of the command on a device visible (**show logging | i ERROR|WARN**) in the output of the test, the following code is used:

any_device.log_user(enable=True)

```
class VerifyLogging(aetest.Testcase):
    <..>
    @aetest.test
    def error_logs(self):
       any_device = self.parent.parameters['dev'][0]
       any_device.log_user(enable=True)
       output = any_device.execute('show logging | i ERROR|WARN')

       if len(output) > 0:
         self.failed('Found ERROR in log, review logs first')
       else:
         pass
```

9. Note that the Setup section of the test case is not used, therefore **pass** is written in this function. We will use the Setup section of the test case later when we execute the **show logging | i ERROR|WARN** command on multiple devices.

```
@aetest.setup
def setup(self):
    pass
```

10. Execute the test script task **71_labpyats.py** and check the results section.

```
$ python3 task71_labpyats.py --testbed pyats_testbed.yaml
```

The Testcase error_log will run only for one device. Scroll above the results section and you will see to which device is related to this output.

```
[2020-01-21 09:42:47,021] +++ asav-1: executing command 'show logging | i ERROR|WARN' +++
show logging | i ERROR|WARN
asav-1#
2020-01-21T09:42:47: %AETEST-INFO: The result of section error_logs is => PASSED
<…>
```

We have learned how to run a testcase for only one device, now we need to get familiar with the **aetest.loop** method, which will let us repeat an elementary test case (written for one device) for every device in the testbed.

11. Open the file **task72_labpyats.py** once again.

```
$ vim task72_labpyats.py
```

12. Pay special attention to the code in **error_logs** method. It receives **device** object as an argument on input and collects the command from this **device**.

```
@aetest.test
def error_logs(self,device):

   output = device.execute('show logging | i ERROR|WARN')

   if len(output) > 0:
    self.failed('Found ERROR in log, review logs first')
   else:
    pass
```

13. Next, check the **setup(self)** method of class **VerifyLogging**. Method **setup(self)** is used to load all the devices from the testbed and to run the **error_logs** method for each device.

```
@aetest.setup
    def setup(self):
        devices = self.parent.parameters['dev']
        aetest.loop.mark(self.error_logs, device=devices)
```

NOTE: aetest.loop.mark() instructs method **self.error_logs** to take an argument for input variable **'device'**, one-by-one from the devices list and run a testcase for each device separately.

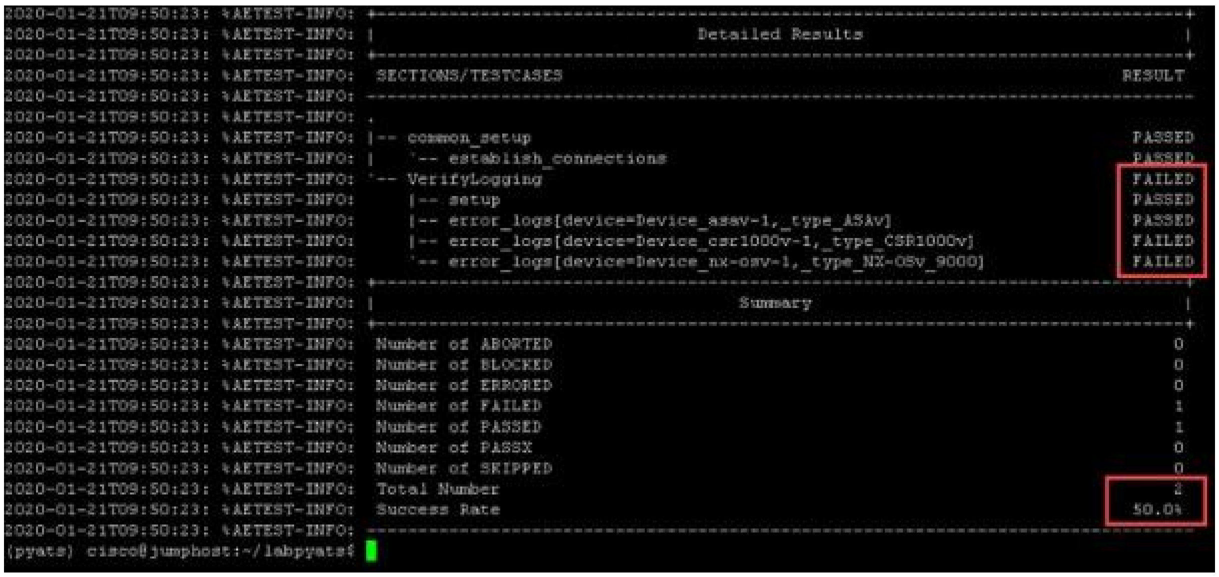
14. Exit Vim without saving.

```
:q!
```

15. Execute the test script; testcase **error_logs** will run for all the devices in the testbed:

```
$ python3 task72_labpyats.py --testbed pyats_testbed.yaml
```

16. Check the **VerifyLogging** results section. The test for ASAv should pass, whereas for CSR1000V and NX-OSv it should fail, since these devices have error messages in the logs.



```
2020-01-21T09:50:23: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T09:50:23: %AETEST-INFO: |                            Detailed Results                                  |
2020-01-21T09:50:23: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T09:50:23: %AETEST-INFO:  SECTIONS/TESTCASES                                                      RESULT
2020-01-21T09:50:23: %AETEST-INFO: --------------------------------------------------------------------------------
2020-01-21T09:50:23: %AETEST-INFO: .
2020-01-21T09:50:23: %AETEST-INFO: |-- common_setup                                                         PASSED
2020-01-21T09:50:23: %AETEST-INFO: |   `-- establish_connections                                            PASSED
2020-01-21T09:50:23: %AETEST-INFO: `-- VerifyLogging                                                        FAILED
2020-01-21T09:50:23: %AETEST-INFO:     |-- setup                                                            PASSED
2020-01-21T09:50:23: %AETEST-INFO:     |-- error_logs[device=Device_asav-1,_type_ASAv]                      PASSED
2020-01-21T09:50:23: %AETEST-INFO:     |-- error_logs[device=Device_csr1000v-1,_type_CSR1000v]              FAILED
2020-01-21T09:50:23: %AETEST-INFO:     `-- error_logs[device=Device_nx-osv-1,_type_NX-OSv_9000]             FAILED
2020-01-21T09:50:23: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T09:50:23: %AETEST-INFO: |                                Summary                                       |
2020-01-21T09:50:23: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T09:50:23: %AETEST-INFO:  Number of ABORTED                                                            0
2020-01-21T09:50:23: %AETEST-INFO:  Number of BLOCKED                                                            0
2020-01-21T09:50:23: %AETEST-INFO:  Number of ERRORED                                                            0
2020-01-21T09:50:23: %AETEST-INFO:  Number of FAILED                                                             1
2020-01-21T09:50:23: %AETEST-INFO:  Number of PASSED                                                             1
2020-01-21T09:50:23: %AETEST-INFO:  Number of PASSX                                                              0
2020-01-21T09:50:23: %AETEST-INFO:  Number of SKIPPED                                                            0
2020-01-21T09:50:23: %AETEST-INFO:  Total Number                                                                 2
2020-01-21T09:50:23: %AETEST-INFO:  Success Rate                                                             50.0%
2020-01-21T09:50:23: %AETEST-INFO: --------------------------------------------------------------------------------
(pyats) cisco@jumphost:~/labpyats$
```

This concludes scenario 6.

Scenario 7. Verify the Service Contracts Coverage

Value Proposition: In this test case, we have the list of devices' serial numbers, covered by the service contracts, and we must verify that all the devices in the testbed are covered by the service contracts. This ensures you will be able to open a TAC case if something goes wrong when the network is in production.

High-level logic of the test will be as follows:

- Connect to each device in the testbed.
- Parse the output of **show inventory** to find the device's serial number (SN).
- Verify whether SN is in the list, covered by the service contracts.

1. Let's use the pyATS shell to check our idea.

```
$ pyats shell --testbed-file pyats_testbed.yaml
```

2. Input the following code into pyATS shell:

```
In [1]: csr = testbed.devices['csr1000v-1']
asa = testbed.devices['asav-1']
nx = testbed.devices['nx-osv-1']
csr.connect()
asa.connect()
nx.connect()
```

3. pyATS uses the Genie **parse** method to collect the output of different show commands and parses it into a structured format (Python dictionary). Let's collect the output of 'show inventory' commands and parse it, using the Genie **parse** method.

```
In [1]: out1 = csr.parse('show inventory')
out2 = asa.parse('show inventory')
out3 = nx.parse('show inventory')
```

4. Now we can observe the structure of parsed outputs. We are starting with the parsed output for CSR1000V. Review it and pay special attention to the highlighted sections.

NOTE: The Python library **pprint** will be imported in this task. This is used to break the output (Python dictionary) onto multiple lines, which is easier to check, instead of having it all on one line.

```
In [1]: import pprint
pprint.pprint(out1)
```

Observe the output:

```
Out [1]: {'main': {'chassis': {'CSR1000V': {'descr': 'Cisco CSR1000V Chassis',
                                   'name': 'Chassis',
                                   'pid': 'CSR1000V',
                                   'sn': '9KZZ4X737UP',
                                   'vid': 'V00'}}},
```

5. Get the serial number of CSR1000V:

```
In [1]: out1['main']['chassis']['CSR1000V']['sn']
```

6. The result of the code should contain a serial number collected in the previous step[1]:

```
Out [1]: '9KZZ4X737UP'
```

7. Obtain the parsed output for ASAv.

```
In [1]: pprint.pprint(out2)
```

Observe the output:

```
Out [1]: {'Chassis': {'description': 'ASAv Adaptive Security Virtual Appliance',
              'pid': 'ASAv',
              'sn': '9ABUANH9G5F',
              'vid': 'V01'}}
```

8. Get the serial number of ASAv:

```
In [1]: out2['Chassis']['sn']
```

9. The result of the code should contain a serial number collected in the previous step[2]:

```
Out [1]: '9ABUANH9G5F'
```

10. Obtain the parsed output for NX-OSv.

```
In [1]: pprint.pprint(out3)
```

Observe the output:

```
Out [1]: {'name': {'Chassis': {'description': 'Nexus9000 9000v Chassis',
                       'pid': 'N9K-9000v',
                       'serial_number': '9712TV4C2JF',
                       'slot': 'None',
                       'vid': 'V02'},
```

11. Get the serial number of NX-OSv:

```
In [1]: out3['name']['Chassis']['serial_number']
```

12. The result of the code should contain a serial number collected in the previous step[2]:

```
Out [1]: '9712TV4C2JF'
```

Now we have all the needed information to write the next test script.

13. Exit the pyATS shell using the **exit** command.

14. Open the file **task8_labpyats.py** in Vim editor:

```
$ vim task8_labpyats.py
```

15. Review the content of the **Inventory** test case, note that we use the data structure learned from pyATS shell in the previous step, to extract a serial number from the output of **show inventory**:

```
@aetest.test
    def inventory(self,device):
        if device.os == 'iosxe':
                 out1 = device.parse('show inventory')
            chassis_sn = out1['main']['chassis']['CSR1000V']['sn']
```

.

[2] Serial number shown below is provided for example and would be different in a real lab.

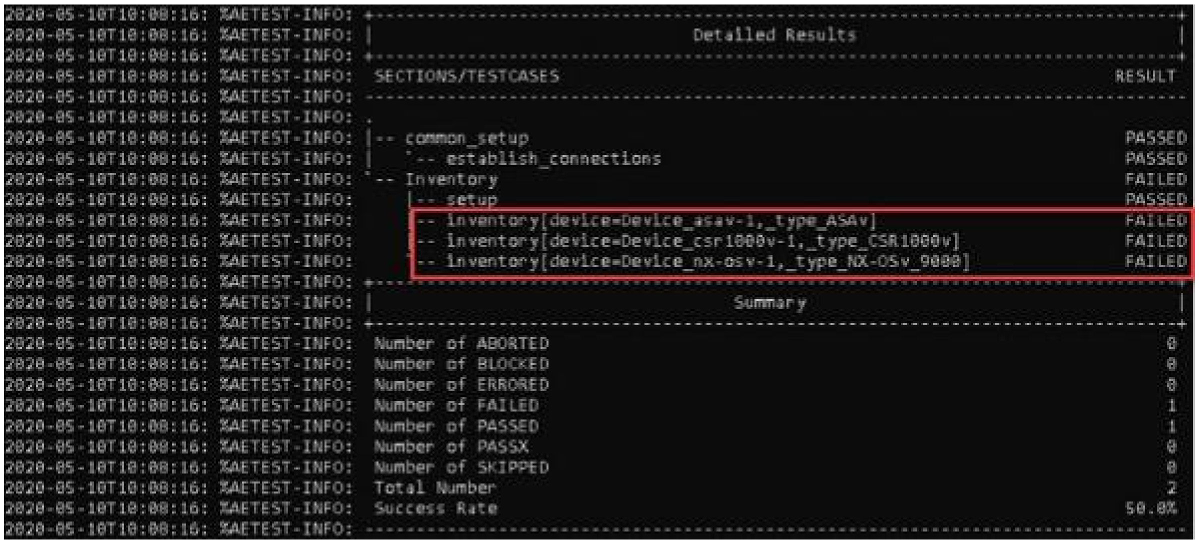
NOTE: The path to fetch the serial number from the structures has been explored in the previous step with pyATS shell. Variables out2 and out3 are used:

```
        elif device.os == 'nxos':
            out2 = device.parse('show inventory')
            chassis_sn = out2['name']['Chassis']['serial_number']
        elif device.os == 'asa':
            out3 = device.parse('show inventory')
            chassis_sn = out3['Chassis']['sn']
```

16. Execute the test script and check the **Detailed Results** section.

```
$ python3 task8_labpyats.py --testbed pyats_testbed.yaml
```

What are the results of these testcases? All fails? Do you have a clue as to why? Continue on for the correct answer.



```
2020-05-10T10:08:16: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-05-10T10:08:16: %AETEST-INFO: |                               Detailed Results                               |
2020-05-10T10:08:16: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-05-10T10:08:16: %AETEST-INFO:  SECTIONS/TESTCASES                                                      RESULT
2020-05-10T10:08:16: %AETEST-INFO: --------------------------------------------------------------------------------
2020-05-10T10:08:16: %AETEST-INFO: .
2020-05-10T10:08:16: %AETEST-INFO: |-- common_setup                                                          PASSED
2020-05-10T10:08:16: %AETEST-INFO: |   `-- establish_connections                                             PASSED
2020-05-10T10:08:16: %AETEST-INFO: `-- Inventory                                                             FAILED
2020-05-10T10:08:16: %AETEST-INFO:     |-- setup                                                             PASSED
2020-05-10T10:08:16: %AETEST-INFO:     |-- inventory[device=Device_asav-1,_type_ASAv]                        FAILED
2020-05-10T10:08:16: %AETEST-INFO:     |-- inventory[device=Device_csr1000v-1,_type_CSR1000v]                FAILED
2020-05-10T10:08:16: %AETEST-INFO:     |-- inventory[device=Device_nx-osv-1,_type_NX-OSv_9000]               FAILED
2020-05-10T10:08:16: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-05-10T10:08:16: %AETEST-INFO: |                                   Summary                                    |
2020-05-10T10:08:16: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-05-10T10:08:16: %AETEST-INFO:  Number of ABORTED                                                            0
2020-05-10T10:08:16: %AETEST-INFO:  Number of BLOCKED                                                            0
2020-05-10T10:08:16: %AETEST-INFO:  Number of ERRORED                                                            0
2020-05-10T10:08:16: %AETEST-INFO:  Number of FAILED                                                             1
2020-05-10T10:08:16: %AETEST-INFO:  Number of PASSED                                                             1
2020-05-10T10:08:16: %AETEST-INFO:  Number of PASSX                                                              0
2020-05-10T10:08:16: %AETEST-INFO:  Number of SKIPPED                                                            0
2020-05-10T10:08:16: %AETEST-INFO:  Total Number                                                                 2
2020-05-10T10:08:16: %AETEST-INFO:  Success Rate                                                             50.0%
2020-05-10T10:08:16: %AETEST-INFO: --------------------------------------------------------------------------------
```

All the tests have failed, since we have serial numbers from a different network in our contract SNs list at the beginning of **task8_labpyats.py** file.

```
contract_sn = ['923C9IN3KU1','93NA29NSARX','9AHA4AWEDBR']
```

17. Replace the serial numbers in the list **contract_sn** with SNs from our testbed's equipment and execute the changed test script once again.

NOTE: Correct SNs from testbed can obtained also from previous script's output:

2020-01-23T13:20:24: %AETEST-ERROR:Failed reason: 9AP535X8NA1 is not covered by contract

<...>

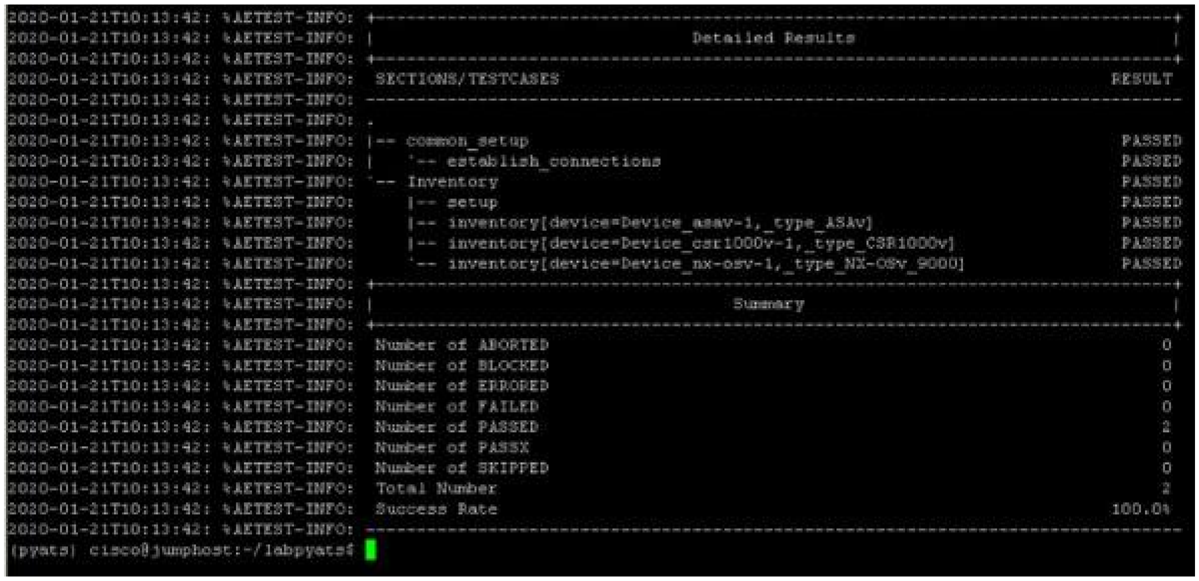
2020-01-23T13:20:25: %AETEST-ERROR:Failed reason: 93HADJOR83W is not covered by contract

<...>

2020-01-23T13:20:26: %AETEST-ERROR:Failed reason: 95B9QWPW1VD is not covered by contract

```
$ python3 task8_labpyats.py --testbed pyats_testbed.yaml
```

Now all the testcases should succeed:



```
2020-01-21T10:13:42: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T10:13:42: %AETEST-INFO: |                            Detailed Results                                  |
2020-01-21T10:13:42: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T10:13:42: %AETEST-INFO:  SECTIONS/TESTCASES                                                      RESULT
2020-01-21T10:13:42: %AETEST-INFO: --------------------------------------------------------------------------------
2020-01-21T10:13:42: %AETEST-INFO: .
2020-01-21T10:13:42: %AETEST-INFO: |-- common_setup                                                         PASSED
2020-01-21T10:13:42: %AETEST-INFO: |   `-- establish_connections                                            PASSED
2020-01-21T10:13:42: %AETEST-INFO: `-- Inventory                                                            PASSED
2020-01-21T10:13:42: %AETEST-INFO:     |-- setup                                                            PASSED
2020-01-21T10:13:42: %AETEST-INFO:     |-- inventory[device=Device_asav-1,_type_ASAv]                       PASSED
2020-01-21T10:13:42: %AETEST-INFO:     |-- inventory[device=Device_csr1000v-1,_type_CSR1000v]               PASSED
2020-01-21T10:13:42: %AETEST-INFO:     `-- inventory[device=Device_nx-osv-1,_type_NX-OSv_9000]              PASSED
2020-01-21T10:13:42: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T10:13:42: %AETEST-INFO: |                               Summary                                        |
2020-01-21T10:13:42: %AETEST-INFO: +------------------------------------------------------------------------------+
2020-01-21T10:13:42: %AETEST-INFO:  Number of ABORTED                                                            0
2020-01-21T10:13:42: %AETEST-INFO:  Number of BLOCKED                                                            0
2020-01-21T10:13:42: %AETEST-INFO:  Number of ERRORED                                                            0
2020-01-21T10:13:42: %AETEST-INFO:  Number of FAILED                                                             0
2020-01-21T10:13:42: %AETEST-INFO:  Number of PASSED                                                             2
2020-01-21T10:13:42: %AETEST-INFO:  Number of PASSX                                                              0
2020-01-21T10:13:42: %AETEST-INFO:  Number of SKIPPED                                                            0
2020-01-21T10:13:42: %AETEST-INFO:  Total Number                                                                 2
2020-01-21T10:13:42: %AETEST-INFO:  Success Rate                                                            100.0%
2020-01-21T10:13:42: %AETEST-INFO: --------------------------------------------------------------------------------
(pyats) cisco@jumphost:~/labpyats$
```

This concludes scenario 7.

Scenario 8. Verify the Routing Information using Parsers and Genie Learn

Value Proposition: In this test case, we have the list of critical routes (usually this is a device's loopback interface) and we must check that these loopbacks are installed in the routing information base (RIB) of all the devices in the testbed.

High-level logic of the test will be as follows:

- Connect to each device in the testbed.
- Learn routing information from RIB of the devices.
- Verify whether all the critical routes are presented in the device's RIB.

1. Let's connect to the pyATS shell and check our idea.

```
$ pyats shell --testbed-file pyats_testbed.yaml
```

2. Input the following code into pyATS shell:

```
In [1]: csr = testbed.devices['csr1000v-1']
asa = testbed.devices['asav-1']
nx = testbed.devices['nx-osv-1']
csr.connect()
asa.connect()
nx.connect()
```

pyATS uses the Genie **learn** method to collect the set of show commands output for a feature configured on the device, to get its snapshot and store it into a structured format (Python dictionary).

```
In [1]: csr_routes = csr.learn('routing')
nx_routes = nx.learn('routing')
```

Now we can observe the structure of the parsed outputs. We are starting with the parsed output for CSR1000V

3. Input the following code into pyATS shell:

```
In [1]: import pprint
pprint.pprint(csr_routes.info)
```

4. Observe the output in pyATS shell:

```
Out [1]:
{'vrf': {'default': {'address_family': {'ipv4': {'routes':

{'10.0.0.12/30': {'active': True,
    'next_hop': {'outgoing_interface': {'GigabitEthernet2':             {'outgoing_interface':
'GigabitEthernet2'}}},
                                                                             'route':
'10.0.0.12/30',

'source_protocol': 'connected',

'source_protocol_codes': 'C'},
'10.0.0.13/32': {'active': True,
                                                                             'next_hop':
{'outgoing_interface': {'GigabitEthernet2': {'outgoing_interface': 'GigabitEthernet2'}}},
                                                                             'route':
'10.0.0.13/32',

'source_protocol': 'local',
'source_protocol_codes': 'L'},
<…>
```

Now we understand that RIB routes for CSR1000V are stored under the following path:

```
In [1]: pprint.pprint(csr_routes.info['vrf']['default']['address_family']['ipv4']['routes'])
```

For NX-OSv, RIB routes are stored under the same path as for CSR1000V:

```
In [1]: pprint.pprint (nx_routes.info['vrf']['default']['address_family']['ipv4']['routes'])
```

5. Exit the pyATS shell using the **exit** command.
6. Open the file **task9_labpyats.py** in Vim editor.

```
$ vim task9_labpyats.py
```

7. Review the content of **routes** testcase, note that we use the path to routes in RIB from the previous step to get the routing information. First we'll get a snapshot of the **routing** feature.

```
@aetest.test
    def routes(self,device):

       if (device.os == 'iosxe') or (device.os == 'nxos'):

         output = device.learn('routing')
         rib = <<replace me>>  # noqa: E999
```

8. Then we compare the loopback routes stored in **golden_routes list**, with the content of **rib**. If the loopback route is not found, then we force the test case to fail.

```
golden_routes = ['192.168.0.3/32','192.168.0.1/32']
<…>
        for route in golden_routes:
           if route not in rib:
             self.failed(f'{route} is not found')
           else:
             pass
Golden routes are /32 networks of loopback interfaces on CSR1000V and NX-OS.
Loopback0 on CSR1000V:
csr1000v-1#sh ip int br
Interface              IP-Address      OK? Method Status                Protocol
GigabitEthernet1       198.18.1.201    YES TFTP   up                    up
GigabitEthernet2       10.0.0.13       YES TFTP   up                    up
GigabitEthernet3       10.0.0.17       YES TFTP   up                    up
Loopback0              192.168.0.3     YES TFTP   up                    up
Loopback0 on NX-OS:
nx-osv-1# sh ip interface brief vrf all

IP Interface Status for VRF "default"(1)
Interface            IP Address      Interface Status
Lo0                  192.168.0.1     protocol-up/link-up/admin-up
Eth1/1               10.0.0.14       protocol-up/link-up/admin-up
Eth1/2               10.0.0.18       protocol-up/link-up/admin-up
Eth1/3               10.0.0.6        protocol-up/link-up/admin-up

IP Interface Status for VRF "management"(2)
Interface            IP Address      Interface Status
```

```
mgmt0                 198.18.1.203    protocol-up/link-up/admin-up

IP Interface Status for VRF "inside"(3)
Interface             IP Address      Interface Status
Lo100                 192.168.100.1   protocol-up/link-up/admin-up
Eth1/4                10.0.0.10       protocol-up/link-up/admin-up
```

9. Complete this test case by replacing the <<replace me>> statement with a rib variable. To accomplish this, you must copy the path to the rib routes, which was explored in the previous step **(output.info['vrf']['default']['address_family']['ipv4']['routes'])**.

Press the **i** button on the keyboard to instruct Vim to start inserting text where at the cursor's location.

10. When you finish, save changes to file **task9_labpyats.py**. Press the **[Esc]** button on your keyboard, then type:

```
:wq
```

11. Execute the test script and check the results section:

```
$ python3 task9_labpyats.py --testbed pyats_testbed.yaml
```

This concludes scenario 8.

Scenario 9. Run PING to Verify Reachability

Value Proposition: In this testcase we must test reachability between devices (NX-OS and CSR1000V), using the ping command.

High-level logic of the test will be as follows:

- Connect to each device in the testbed.
- Find links between NX-OS and CSR1000V.
- Collect IP addresses from both ends of these links.
- Run the ping commands from NX-OS and CSR1000V, for IP addresses, discovered in the previous step.

1. Let's connect to pyATS shell and check our idea:

```
$ pyats shell --testbed-file pyats_testbed.yaml
```

2. Input the following code into pyATS shell:

```
In [1]: csr = testbed.devices['csr1000v-1']
nx = testbed.devices['nx-osv-1']
```

3. pyATS has a **find_links(device_name)** method to find all the links between two devices in the topology. Let's find the links between CSR1000V and NX-OSv.

```
In [1]: nx.find_links(csr)
```

4. Observe the output:

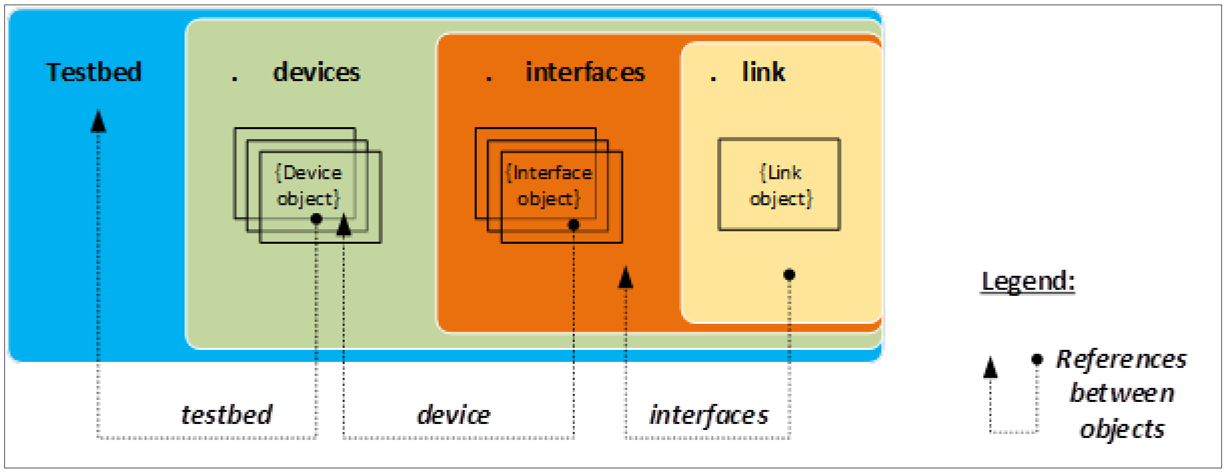
```
Out [1]: {<Link object 'csr1000v-1-to-nx-osv-1' at 0x7f445194b050>,
 <Link object 'csr1000v-1-to-nx-osv-1#1' at 0x7f445194b150>,
 <Link object 'flat' at 0x7f445194b410>}
```

5. Exit the pyATS shell using the **exit** command.

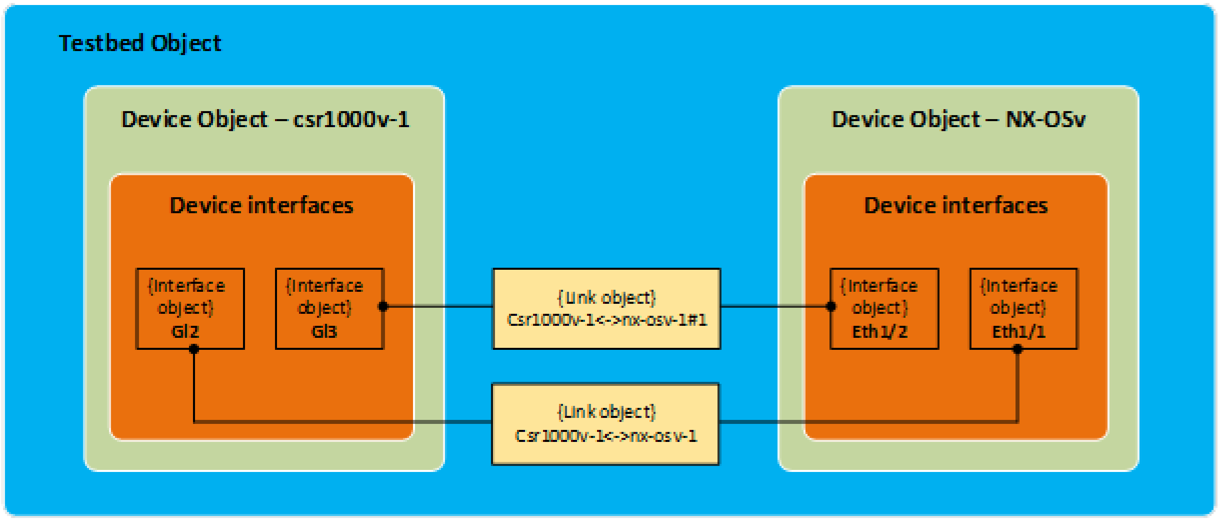
Before studying the code and running the next script, let's dive into the details on how information about a topology is stored in a testbed object (see the illustration that follows for a graphical representation of the explanation).

Things to know about the structure of the testbed object (created from the testbed YAML file specified: **testbed.yaml**):

- The pyATS **Testbed** object contains the Python dictionary **devices.**
- Elements of the **devices** dictionary are the **Device** objects.
- Each object in the **devices** dictionary stores dictionary **interfaces** (contains **interface** objects).
- Each **interface** object stores the **link** object.



- The **Testbed** object is the top container object, containing all the testbed devices and all the subsequent information that is generic to the testbed.
- **Device** objects represent physical and/or virtual hardware in a testbed topology.
- **Interface** objects represent a physical/virtual interface that connects to a link of some sort (for example, Ethernet, ATM, Loopback, and so on).
- **Link** objects represent the connection (wire) between two or more interfaces within a testbed topology.


Testbed Object
Device Object – csr1000v-1
Device Object – NX-OSv
Device interfaces
{Interface object} Gi2
{Interface object} Gi3
{Link object} Csr1000v-1<->nx-osv-1#1
{Link object} Csr1000v-1<->nx-osv-1
{Interface object} Eth1/2
{Interface object} Eth1/1

Let's check the structure depicted above using our topology. We will find all the links connected between **nx-osv-1** and **csr1000v-1**.

IMPORTANT: We can get the value of an attribute for each object. For example, we can get a link to which an interface object belongs by calling a **link** attribute. We can also reference interfaces which belong to this link, by calling the **interfaces** attribute in step 6 (see code below).

```
$ pyats shell --testbed-file pyats_testbed.yaml
```

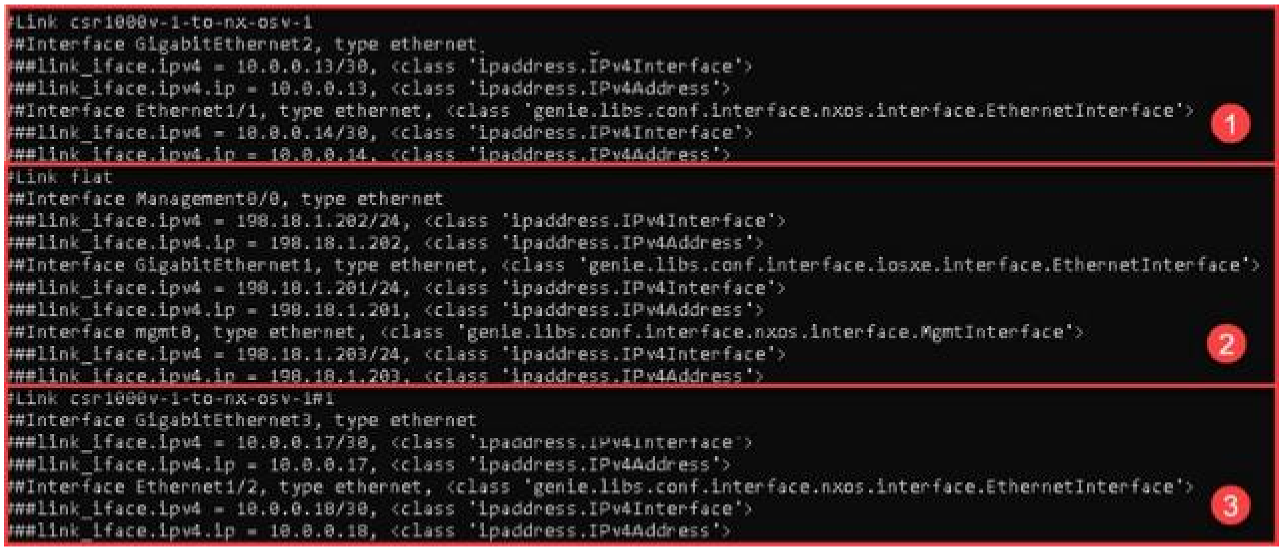
6. Input the following code into pyATS shell:

NOTE: Because this lab guide is in PDF format, it doesn't allow the use of copy and paste for proper indentation in the pyATS interactive shell. To copy and paste the code snippet below, use the appropriate section in the **cisco_automated_testing-24072020.txt** file provided with this lab. A straight copy and paste from the PDF will not work.

```
In [1]: csr = testbed.devices['csr1000v-1']
nx = testbed.devices['nx-osv-1']
links = nx.find_links(csr)

for link in links:
print(f'#{link}')
for link_iface in link.interfaces:
print(f'##{link_iface}')
print(f'###link_iface.ipv4 = {link_iface.ipv4}, {type(link_iface.ipv4)}')
print(f'###link_iface.ipv4.ip = {link_iface.ipv4.ip}, {type(link_iface.ipv4.ip)}')
```

Refer to the command output:

- **#Link csr1000v-1-to-nx-osv-1** – represents interfaces of all devices connected to the first link between **csr1000v-1** and **nx-osv-1**.
- **#Link flat** – represents interfaces of all devices (**asav-1**, **csr1000v-1**, **nx-osv-1**) connected to a management network.
- **#Link csr1000v-1-to-nx-osv-1**#1 - represents interfaces of all devices connected to the second link between **csr1000v-1** and **nx-osv-1**.



```
#Link csr1000v-1-to-nx-osv-1
##Interface GigabitEthernet2, type ethernet
###link_iface.ipv4 = 10.0.0.13/30, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 10.0.0.13, <class 'ipaddress.IPv4Address'>
##Interface Ethernet1/1, type ethernet, <class 'genie.libs.conf.interface.nxos.interface.EthernetInterface'>
###link_iface.ipv4 = 10.0.0.14/30, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 10.0.0.14, <class 'ipaddress.IPv4Address'>
#Link flat
##Interface Management0/0, type ethernet
###link_iface.ipv4 = 198.18.1.202/24, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 198.18.1.202, <class 'ipaddress.IPv4Address'>
##Interface GigabitEthernet1, type ethernet, <class 'genie.libs.conf.interface.iosxe.interface.EthernetInterface'>
###link_iface.ipv4 = 198.18.1.201/24, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 198.18.1.201, <class 'ipaddress.IPv4Address'>
##Interface mgmt0, type ethernet, <class 'genie.libs.conf.interface.nxos.interface.MgmtInterface'>
###link_iface.ipv4 = 198.18.1.203/24, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 198.18.1.203, <class 'ipaddress.IPv4Address'>
#Link csr1000v-1-to-nx-osv-1#1
##Interface GigabitEthernet3, type ethernet
###link_iface.ipv4 = 10.0.0.17/30, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 10.0.0.17, <class 'ipaddress.IPv4Address'>
##Interface Ethernet1/2, type ethernet, <class 'genie.libs.conf.interface.nxos.interface.EthernetInterface'>
###link_iface.ipv4 = 10.0.0.18/30, <class 'ipaddress.IPv4Interface'>
###link_iface.ipv4.ip = 10.0.0.18, <class 'ipaddress.IPv4Address'>
```

7. Open the file **task10_labpyats.py** in Vim editor:

```
$ vim task10_labpyats.py
```

8. Review the content of the **PingTestcase** test case, look at the **def setup(self)** function. Code in this function follows the logic used in the previous step:
   - Get all the links between NX-OSv and CSR1000V (**nx.find_links(csr)**).
   - Get interfaces for each link (**for iface in links.interfaces**) and append its IPv4 address (**iface.ipv4.ip**) into list **dest_ips**, to use them further in **ping** commands.

To exclude management IP addressing space, there is a check whether an IP address on a link is from a management address space (if **dest_ip** not in **mgmt_net**). In the event that an IP address is from a management IP address, it's not appended to the list **dest_ips**.

NOTE: Note that the IP address in the **link_iface.ipv4.ip** object is of the IPv4Address type so we can check whether it overlaps with IPv4Network without any conversion of type (hence **if dest_ip not in mgmt_net** is used).

The code of the **setup(self)** function is shown below:

```
mgmt_net = IPv4Network('198.18.1.0/24')

# Find links between NX-OS device and CSR1000v
for link in links:
    # process each link between devices

    for link_iface in link.interfaces:
        # process each interface (side) of the link and extract IP address from it

        dest_ip = link_iface.ipv4.ip

        # Check that destination IP is not from management IP range
        if dest_ip not in mgmt_net:
            log.info(f'{link_iface.name}:{link_iface.ipv4.ip}')
            dest_ips.append(link_iface.ipv4.ip)
        else:
            log.info(f'Skipping link_iface {link_iface.name} from management subnet')
```

In function **def ping(self, dest_ip)** we execute a **ping** command for each IPv4 address of both ends of the links between NX-OSv and CSR1000V.

NOTE: Note the following:

1. In this task we are not passing **Device** objects into **@aetest.test** from **@aetest.setup** using **aetest.loop.mark** as it has been done in previous tasks:

aetest.loop.mark(self.error_logs, device=devices)

2. In this task we are passing **dest_ip** one-by-one from the **dest_ips** list:

aetest.loop.mark(self.ping, dest_ip=dest_ips)

4. To get a Device object we call the **self.parent.parameters** attribute:

nx = self.parent.parameters['testbed'].devices['nx-osv-1']

The string returned by a ping operation is shown below. The field that must be extracted is marked yellow:

```
5 packets transmitted, 5 packets received, 0.00% packet loss
```

To check this field, we use a regular expression, it extracts packet loss from the ping command's output. If the loss rate is less than 20% (to accommodate the potential first ping drop due to ARP resolution) then the test case should pass successfully:

```
nx = self.parent.parameters['testbed'].devices['nx-osv-1']

try:
   result = nx.ping(dest_ip)
<…>
else:
    m = re.search(r"(?P<rate>\d+)\.\d+% packet loss", result)
    loss_rate = m.group('rate')

    if int(loss_rate) < 20:
        self.passed(f'Ping loss rate {loss_rate}%')
    else:
        self.failed('Ping loss rate {loss_rate}%')
```

9. Execute the created test script and check the **results** section; all pings should succeed:

```
$ python3 task10_labpyats.py --testbed pyats_testbed.yaml
```

This concludes scenario 9.

Scenario 10. Show the Results of Tests in a Browser

Value Proposition: In this last task, we will see how to show the results of the tests in a more user-friendly way in a browser. For this, the **pyats run job** command will be used in a Bash shell.

When a test is run using pyats run job it adds the following advantages:

- Logs of test runs are saved into the archive
- Graphical representation of test results in a browser
- Ability to run tests in different Python scripts

To use **pyats run job**, a special file "job file" (written in Python) should be created.

A job file looks as shown below:

- **<test_name1>** - specifies the path in the system to the Python file with the first list of tests (for example **task9_labpyats.py**).
- **<test_name2>** - specifies the path to the Python file with the second list of tests (for example **task10_labpyats.py**).

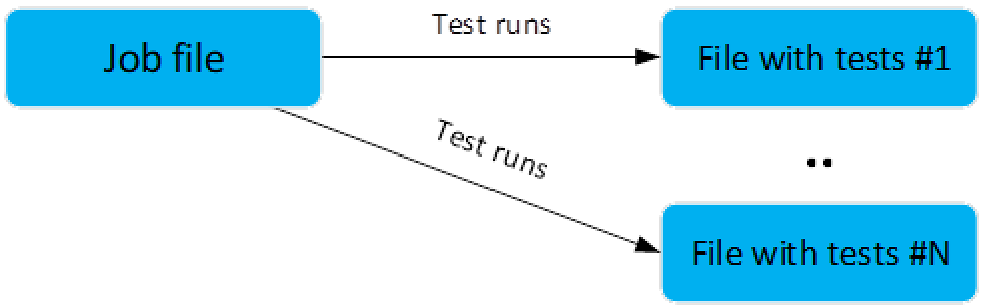
The method **run** from the imported library **ats.easypy** instructs the system to run tests in sequence.

```
import os
from ats.easypy import run

run(testscript=<test_name1>)
run(testscript=<test_name2>)
import os
from ats.easypy import run

def main():
    # Find the location of the script in relation to the job file
    <test_name1> = os.path.join('<file_with_tests1.py>')
    <test_name2> = os.path.join('<file_with_tests2.py>')

    # Execute the testscript
    run(testscript=<test_name1>)
    run(testscript=<test_name2>)
```

In order to call **pyats run job**, use the following command in a Bash shell:

```
$ pyats run job <job-file> --testbed <testbed-file>
```

Schematically, the process of **pyats run job** can be shown as follows:



Let’s use **pyats job run** to execute tests from Task10. PyATS job file **task11_runtestsjob.py** has been pre-configured for this.

1. Open **runtestsjob.py** file in Vim and check it (the structure is in accordance to the one shown above).

```
$ vim task11_runtestsjob.py
```

2. Execute the pyATS job file with the **pyats run job** command:

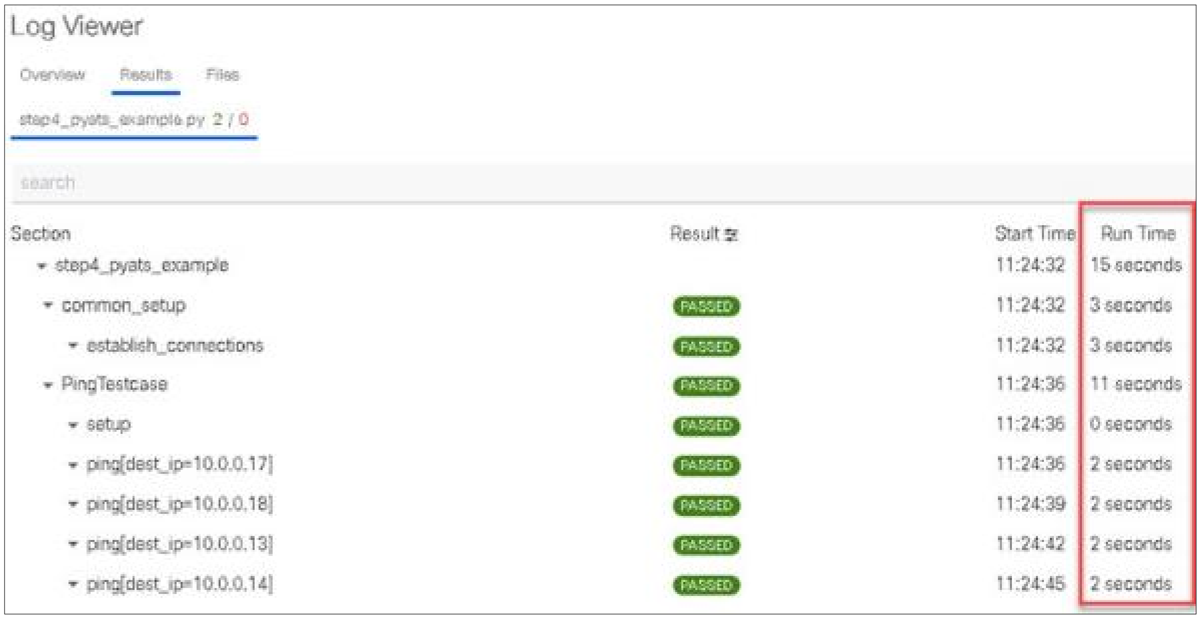
```
$ pyats run job task11_runtestsjob.py --testbed pyats_testbed.yaml
```

3. After completion of the test, check the results:

```
$ pyats logs view
```

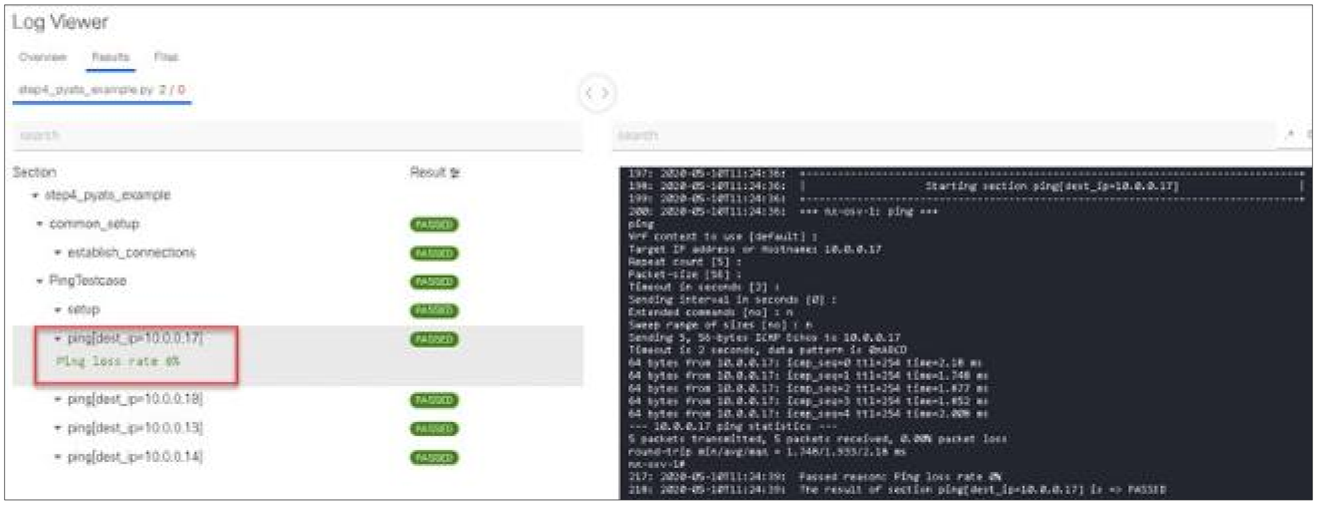
4. Google Chrome would be opened to show results (see the next illustration).

NOTE: Pay special attention to the result of each test, which is shown along with start time and run time of each test.



Section	Result	Start Time	Run Time
step4_pyats_example		11:24:32	15 seconds
common_setup	PASSED	11:24:32	3 seconds
establish_connections	PASSED	11:24:32	3 seconds
PingTestcase	PASSED	11:24:36	11 seconds
setup	PASSED	11:24:36	0 seconds
ping[dest_ip=10.0.0.17]	PASSED	11:24:36	2 seconds
ping[dest_ip=10.0.0.18]	PASSED	11:24:39	2 seconds
ping[dest_ip=10.0.0.13]	PASSED	11:24:42	2 seconds
ping[dest_ip=10.0.0.14]	PASSED	11:24:45	2 seconds

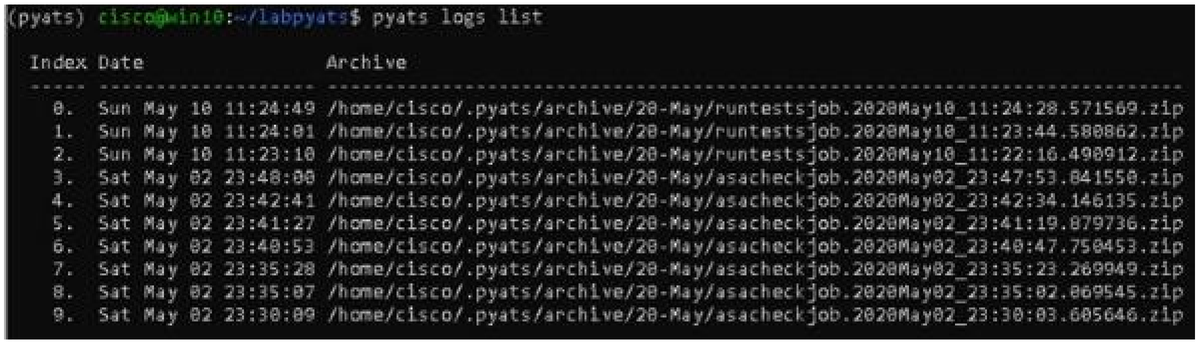
5. Click on the test **ping[dest_ip=10.0.0.17]**, a detailed log from the execution of this test will be shown on the right side of the window.



```
197: 2020-05-10T11:24:36: +------------------------------------------------------------------------------+
198: 2020-05-10T11:24:36: |                   Starting section ping[dest_ip=10.0.0.17]                   |
199: 2020-05-10T11:24:36: +------------------------------------------------------------------------------+
200: 2020-05-10T11:24:36: +++ nx-osv-1: ping +++
ping
VRF context to use [default] :
Target IP address or Hostname: 10.0.0.17
Repeat count [5] :
Packet-size [56] :
Timeout in seconds [2] :
Sending interval in seconds [0] :
Extended commands [no] : n
Sweep range of sizes [no] : n
Sending 5, 56-bytes ICMP Echos to 10.0.0.17
Timeout is 2 seconds, data pattern is 0xABCD
64 bytes from 10.0.0.17: icmp_seq=0 ttl=254 time=2.18 ms
64 bytes from 10.0.0.17: icmp_seq=1 ttl=254 time=1.748 ms
64 bytes from 10.0.0.17: icmp_seq=2 ttl=254 time=1.877 ms
64 bytes from 10.0.0.17: icmp_seq=3 ttl=254 time=1.852 ms
64 bytes from 10.0.0.17: icmp_seq=4 ttl=254 time=2.008 ms
--- 10.0.0.17 ping statistics ---
5 packets transmitted, 5 packets received, 0.00% packet loss
round-trip min/avg/max = 1.748/1.933/2.18 ms
nx-osv-1#
217: 2020-05-10T11:24:39:  Passed reason: Ping loss rate 0%
218: 2020-05-10T11:24:39:  The result of section ping[dest_ip=10.0.0.17] is => PASSED
```

6. Stop the Log Viewer by pressing **CTRL+C** in the Bash shell. Check the list of the logs in an archive (each execution of **pyats run job** is depicted in a separate line):

```
$ pyats logs list
```



```
(pyats) cisco@win10:~/labpyats$ pyats logs list

 Index Date                 Archive
 ----- -------------------- ------------------------------------------------------------------------------
    0.  Sun May 10 11:24:49 /home/cisco/.pyats/archive/20-May/runtestsjob.2020May10_11:24:28.571569.zip
    1.  Sun May 10 11:24:01 /home/cisco/.pyats/archive/20-May/runtestsjob.2020May10_11:23:44.580862.zip
    2.  Sun May 10 11:23:10 /home/cisco/.pyats/archive/20-May/runtestsjob.2020May10_11:22:16.490912.zip
    3.  Sat May 02 23:48:00 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:47:53.041550.zip
    4.  Sat May 02 23:42:41 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:42:34.146135.zip
    5.  Sat May 02 23:41:27 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:41:19.079736.zip
    6.  Sat May 02 23:40:53 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:40:47.750453.zip
    7.  Sat May 02 23:35:28 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:35:23.269949.zip
    8.  Sat May 02 23:35:07 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:35:02.069545.zip
    9.  Sat May 02 23:30:09 /home/cisco/.pyats/archive/20-May/asacheckjob.2020May02_23:30:03.605646.zip
```

7. Execute **pyats run job** two more times.

```
$ pyats run job task11_runtestsjob.py --testbed pyats_testbed.yaml
```

8. Check the list of the logs in an archive (new lines should be added each signifying **pyats run job** was executed in the previous step).

9. Open the 2nd report from the list in a web browser:

```
$ pyats logs view -2
```

pyATS run is a very handy tool and it's recommended that you use it to run your pyATS tests.

You might also check the official documentation for the details at this site:

https://pubhub.devnetcloud.com/media/pyats/docs/cli/pyats_run.html#pyats-run-job

This concludes scenario 10.

Conclusion

You have learned how to build your automated tests using the pyATS framework.

These tools provide a wide variety of opportunities, and it’s not hard to start using them.

This lab has introduced you to real-world examples and, we hope, has given you a head start on the automation of tests in your network.

The main points we wanted to emphasize in this lab include:

- Test automation for network operations is available today.
- It's easy to implement automation in your network, with little to no programming experience.
- pyATS is a rather simple and extensible framework for automation.

What's Next?

Install pyATS:

1. Download a Docker image with pyATS:

https://github.com/CiscoTestAutomation/pyats-docker

2. Or, install pyATS on your Linux/macOS. Note that pyATS is supported on Windows only via WSL.

https://developer.cisco.com/docs/pyats-getting-started/

3. Use the following information for a quick start with pyATS:

https://github.com/CiscoTestAutomation/getting-started/tree/master/start-guide

Code used in the lab tasks

Link to the repository with code used in this lab:

https://github.com/cleur2293/labpyats