LABOPS-1775

Automated testing for your network **(pyATS/Genie)**

Sergey Sazhin, Technical Leader, CX
Alexey Sazhin, Technical Leader, CX
Vladimir Savostin, Technical Leader, CX
Munther Kojeh, Customer Support Engineer

Learning Objectives

During this lab you will get hands-on experience with PyATS and Genie which are vendor-agnostic suite of libraries for Python.

You will learn how to build your own automated tests, using these frameworks.
PyATS and Genie open wide opportunities ("sky is the limit") 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 smoothly start implementing automation of tests in your network.

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

- Build testbed from VIRL[1] topology
- Understand pyATS/Genie suite capabilities
- Start writing automated tests for your network

Disclaimer

In this lab we want to demonstrate capabilities of pyATS/Genie, and since this session is introductory and does not require prior programming experience, we made our examples rather comprehensive, to provide you with clear understanding of the pyATS/Genie concept, however, not too elaborate and meant for distribution in real networks.

[1] VIRL – Virtual Internet Routing Lab.

Table of contents

Get started

Step 1 Anyconnect VPN verification

Verify that you are connected to the lab using Cisco AnyConnect:

1. Open Window's System Tray icons.
2. Click on Cisco AnyConnect icon.
3. Verify that you are connected to host: dcloud-rtp-anyconnect.cisco.com.

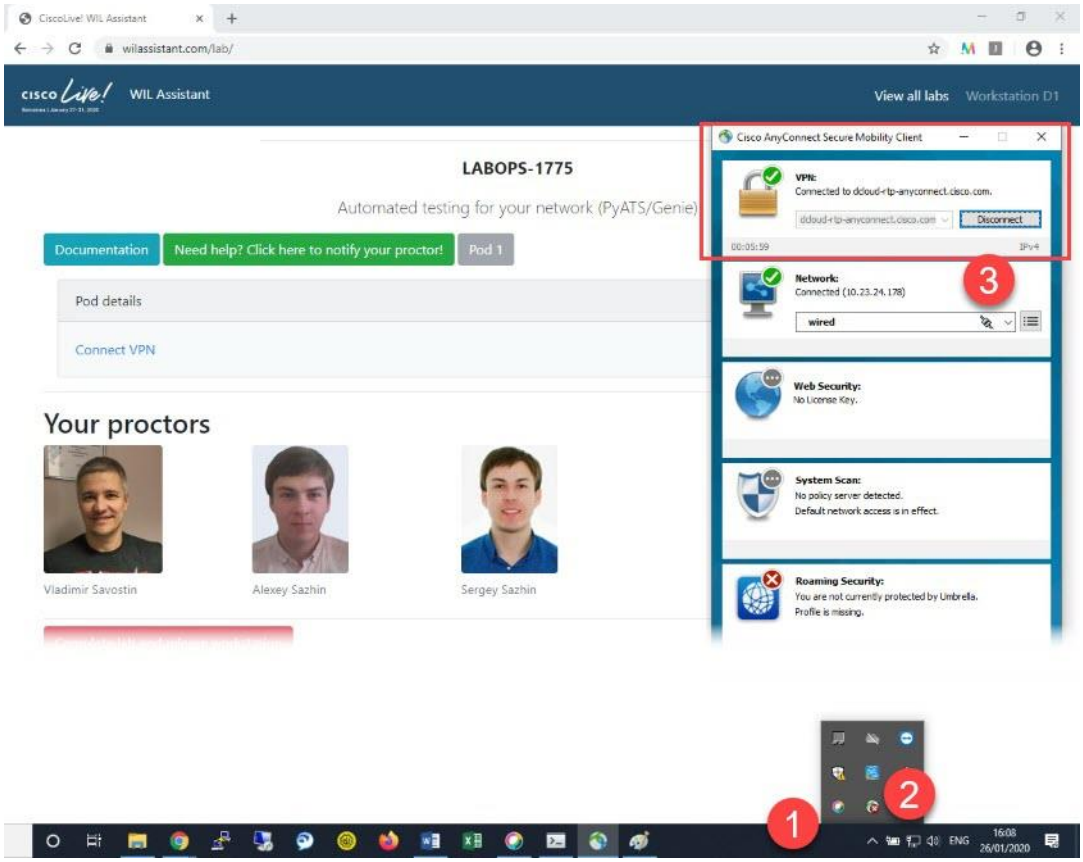
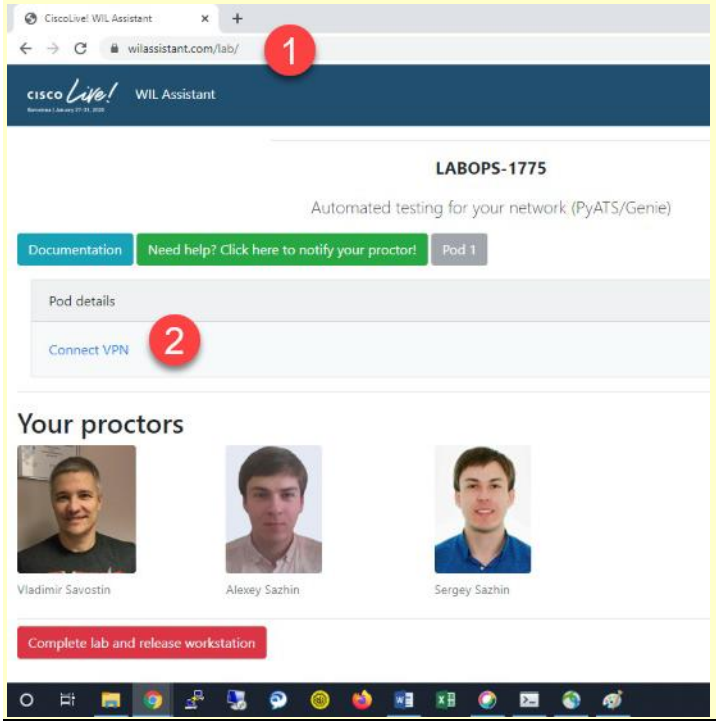


Figure 1 – Anyconnect VPN verification

If you see that you are not connected to POD's Anyconnect VPN, then return to web page www.wilassistant.com and click Connect VPN hyperlink once again:



Step 2 RDP connection to jumphost

Establish RDP connection, using Microsoft Remote Dektop Client[2], using following credentials:

RDP IP address:	198.18.133.252
RDP credentials:	User: **Administrator** Password: **C1sco12345**

[2] Press Start and put mstsc in search field to find Microsoft Remote Desktop application.

Scenario

Disclaimer:

This is a work of fiction. Names, characters, businesses, places, events, locales, and incidents are either the products of the author's imagination or used in a fictitious manner. Any resemblance to actual persons, living or dead, or actual events is purely coincidental. But it's based on a true story.

Imagine the following situation is happening:
You are a network engineer who is implementing a project for a small Datacenter. You expect this to be the standard project, that might repeat in the future. Based on this, you have decided to perform tests from your "Network Ready for Use" document programmatically. This approach will help you to save time for similar projects in the future.

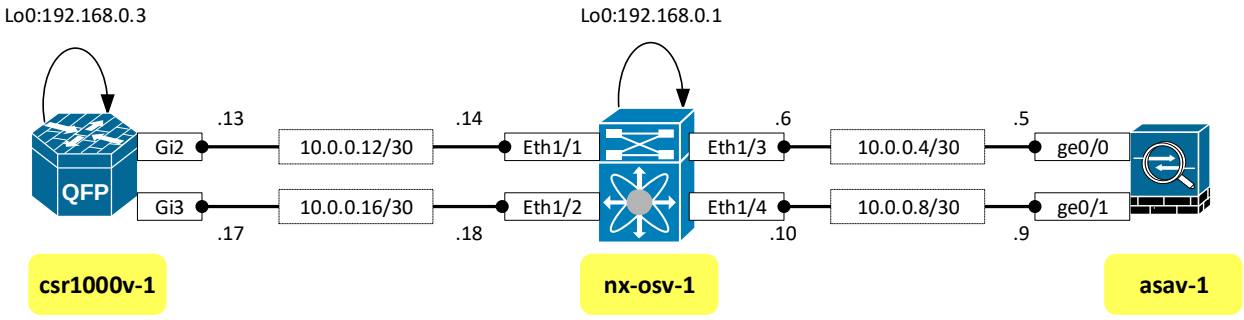


Figure 2 – Lab diagram

Linux cheatsheet

```
# Go to the home directory
cd ~/

# Change to the labpyats sub-directory
cd labpyats

# View the contents of file at once
cat <filename>

# View the contents of file, display one screen at a time if file is
large
more <filename>

# Open file in Vim text editor, for editing
vim <filename>

# List all files in a directory
ls -l

# Display current working directory
pwd
```

Vim editor cheatsheet:

```
i - insert in the cursor
Esc - exit insert mode
u - undo
:wq - write (save) and quit
:q! - quit without saving
```

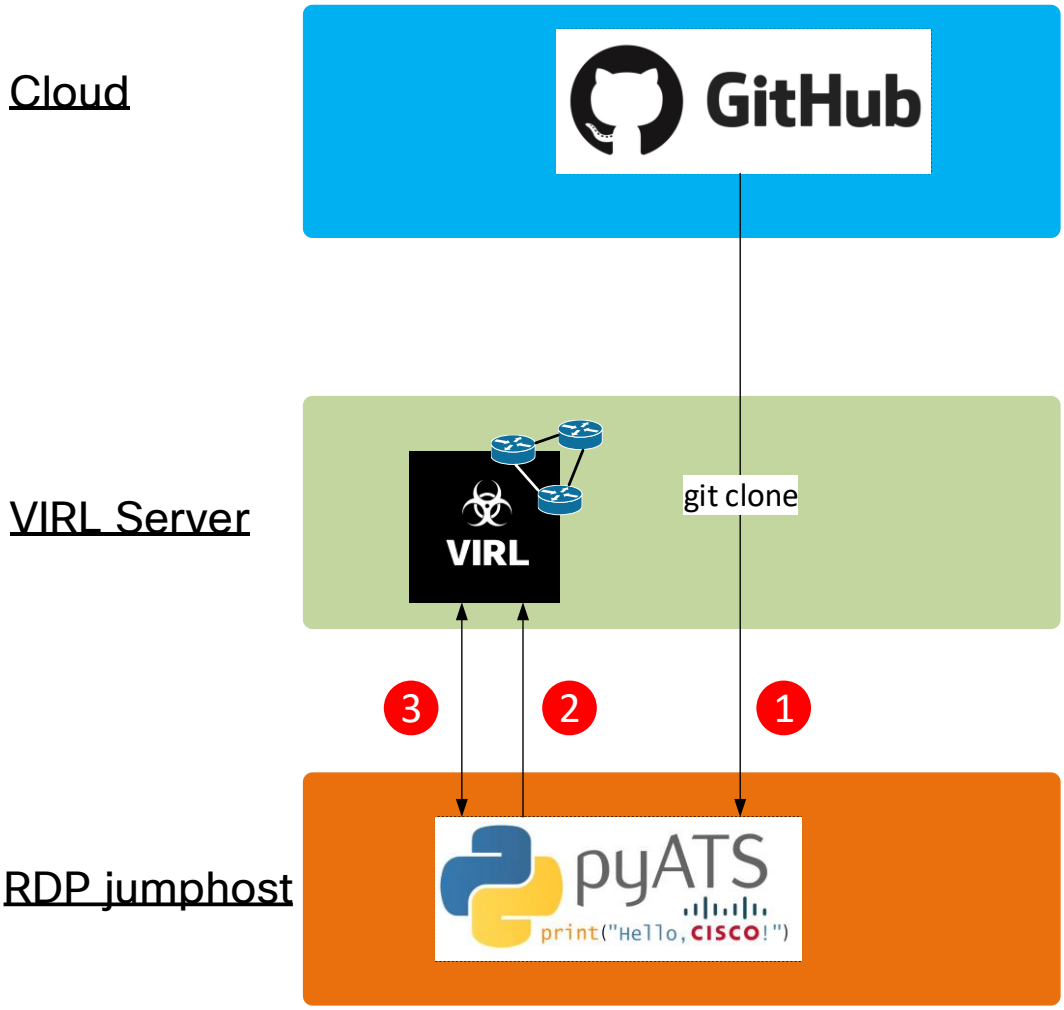


Figure 3 – High-level Lab's Topology

Here are main components of this lab, depicted on the picture above:

- GitHub – used to store the code of our lab in the Internet and make it available for download (git clone) to pyATS.
- VIRL Server – emulates network devices and topology.
- PyATS – vendor-agnostic suite of libraries for Python, they will be used to build and run tests on network topology, emulated on VIRL Server.

Lab's task sequence will be as follows:

1. Clone project from GitHub to RDP jumphost.
2. Start emulated network topology on VIRL Server.
3. Run tests from pyATS on emulated network topology.

In your real-world network VIRL emulated topology can be represented either by your lab setup or production network that should be tested.

Before we begin

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

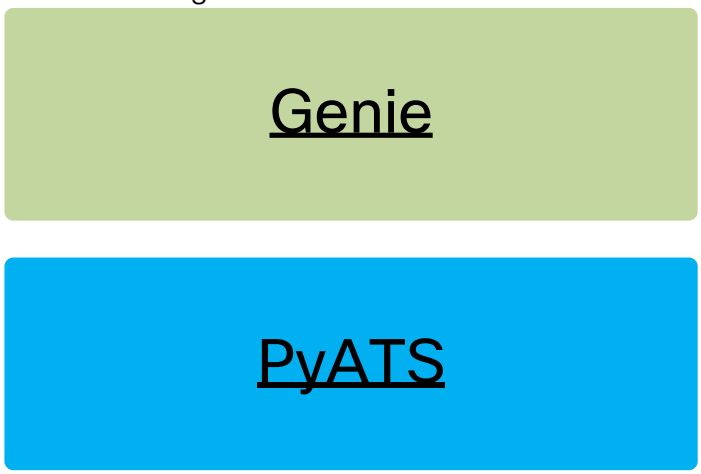


Figure 4 – PyATS and Genie functionality

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

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

Genie is pyATS's library which is used to process data which is collected from networking devices, by leveraging:

- parsers: converting/formatting command output into Python data structures
- learn feature: to collect protocol configuration state and operational status of device and put it into Python data structures.

Task 1: Getting started

1.1. Click on window with RDP session to RDP jumphost

Rest of the lab steps are done from RDP connection to jumphost

1.2. Run Ubuntu 18.04 by clicking respective shortcut on desktop of jumphost:

Ubuntu Linux will run on our RDP jumphost, on top of Windows 10[3].

[3] Windows Subsystem for Linux (WSL), non-beta support starting from Windows 10 Fall Creators Update (version 1709).
Installation instructions:

1.3. Bash (Linux) shell would be opened:

1.4. Activate virtual environment by running the following command:

```
$ source /home/cisco/pyats/bin/activate
```

Throughout the lab you will be working from virtual environment[4].

Virtual environment provides the following major advantages over running Python scripts globally:

1. Project isolation: Avoid installing Python packages globally which could break system tools or other projects.

2. Dependency management: Make the project self-contained and reproducible by capturing all package dependencies in a requirements file.

Virtual environment has been pre-installed in the following directory (from which it's activated):

/home/cisco/pyats

Ensure you have started working in the virtual environment "pyats".
Keyword (pyats) in the beginning of each line would signify this:

```
(pyats) cisco@win10$
```

1.5. Clone the project for this lab from Github repository:

Git clone should be issued from home directory, hence we changing our working directory to home (cd ~/)

```
cd ~/
git clone https://github.com/cleur2293/labpyats.git
(pyats) cisco@win10$ cd labpyats/
```

1.6. Check the project's structure:

```
ls -l
```

Note, this section is for information only, no action and special attention is required.

Filename	Description	Task #
empty_pyats_example.py	Initial task for pyATS exploration	Task 4
topology.virl	Network topology for VIRL	Task 1

https://docs.microsoft.com/en-us/windows/wsl/install-win10

[4] If you need to exit virtual environment, use deactivate command in bash shell.

README.md	Readme file	-
step0_pyats_example.py	Task for collection of show commands	Task 3
step1_pyats_example.py	Test to verify log messages	Task 5
step2_pyats_example.py	Test to verify service contracts coverage	Task 6
step3_pyats_example.py	Test to verify routing information	Task 7
step4_pyats_example.py	Test to verify reachability (ping)	Task 8

Task 2: Convert VIRL topology to pyATS testbed file

There is a VIRL topology file for this lab pre-created. The network depicted on Figure 2 - Lab diagram is defined in this file. The name of the VIRL topology file: topology.virl

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, it would be required to convert topology.virl file to testbed YAML file, which is recognized by pyATS. Fortunately, virlutils Python package5 can make this conversion for us!

2.1. Ensure there is all the information needed to connect to VIRL server. Access credentials for VRL are pre-configured in .virlrc file stored in home directory:

```
$ cat ~/.virlrc
VIRL_HOST=virl.demo.dcloud.cisco.com
VIRL_USERNAME=guest
VIRL_PASSWORD=guest
```

2.2. Ensure you are in /home/cisco/labpyats directory[6]:

```
$ pwd
/home/cisco/labpyats
```

2.3. Start VIRL simulation (topology.virl), using virlutils (to run virlutils use virl keyword in bash shell):

```
$ virl up
Creating default environment from topology.virl
```

2.4. Ensure the start of VIRL topology has been initiated (in “Status” column “ACTIVE” state is shown):

[5] It has been pre-installed from PyPi using pip:
pip install virlutils.
[6] Further steps (2.3 - 2.7) involving use of virlutils are to be done from /home/cisco/labpyats directory.

```
$ virl ls
```

Example of running VIRL topology:

```
(pyats) cisco@win10:~/labpyats$ virl ls
Running Simulations
```

Simulation	Status	Launched	Expires
labpyats_default_iurjDh	ACTIVE	2020-01-20T12:49:27.490928	

Figure 5 – Example of running VIRL topology

2.5. Wait for ~5 minutes for all the devices to become reachable (in "Reachable" column "REACHABLE" state is shown).
You can check it using the following command: $ virl nodes

Here is a list of all the running nodes

Node	Type	State	Reachable	Protocol	Management Address	External Address
asav-1	ASAv	ACTIVE	REACHABLE	telnet	198.18.1.202	N/A
csr1000v-1	CSR1000v	ACTIVE	REACHABLE	telnet	198.18.1.201	N/A
jumphost	server	ACTIVE	REACHABLE	ssh	198.18.1.100	N/A
nx-osv-1	NX-OSv 9000	ACTIVE	REACHABLE	telnet	198.18.1.203	N/A
~mgmt-lxc	mgmt-lxc	ACTIVE	REACHABLE	ssh	198.18.1.171	198.18.1.172

Figure 6 – Example of all nodes in VIRL topology reachable

2.6. After all the devices have become reachable, export running VIRL topology to YAML file[7]: 'pyats_testbed.yaml'.

```
$ virl generate pyats -o pyats_testbed.yaml
Writing pyats_testbed.yaml
```

2.7. Observe the content of created pyats_testbed.yaml file.

```
$ more pyats_testbed.yaml
testbed:

  name: labpyats_default_WPSKJx <-- name of topology simulation

  tacacs: <-- credentials for CLI access
    username: "%ENV{PYATS_USERNAME}"
  passwords:
    tacacs: "%ENV{PYATS_PASSWORD}"
    enable: "%ENV{PYATS_AUTH_PASS}"
    line: "%ENV{PYATS_PASSWORD}"
<…>
```

[7] This YAML testbed file is to be further used by PyATS.

```
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[8] is Python package and used by
pyATS to connect to network devices through command-line interface.
      console:
        protocol: telnet
        ip: virl.demo.dcloud.cisco.com
        port: 17000 <-- connection to device would be done via console
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 there is all the required information to start our tests with pyATS.

Note that username and passwords to access devices are not stored in yaml file:

```
  username: "%ENV{PYATS_USERNAME}"

passwords:

  tacacs: "%ENV{PYATS_PASSWORD}"

  enable: "%ENV{PYATS_AUTH_PASS}"
```

It's recommended to store credentials separately, as environmental variables.

2.8. In the lab these environment variables are stored in ~/.bashrc file. Check these environment variables are specified in this file:

[8] See more about Unicon connection library at PyPI: https://pypi.org/project/unicon/

```
$ tail -n 4  ~/.bashrc
export PYATS_USERNAME=cisco
export PYATS_PASSWORD=cisco
export PYATS_AUTH_PASS=cisco
export PYATS_PASSWORD=cisco
```

2.9. Check contents of these environment variables from bash shell:

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

Task 3: Observe pyATS capabilities, using ipyats

We believe, it's always faster to start learning of pyATS in interactive format, when you can try different pyATS functions. It allows 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 interactive way of developing tests: "interactive pyATS" shell or simply **ipyats**[9].

Let's start our journey with ipyats, using it with pyats_testbed.yaml, created in the previous chapter.

3.1. Run ipyats from the bash shell, specify YAML testbed file as the parameter:

```
$ ipyats --testbed pyats_testbed.yaml
```

Interactive shell would be opened, where you can input the Python code:

```
    Welcome to ipyATS!
<..>
Python 3.7.5 (default, Nov  7 2019, 10:50:52)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.11.1 -- An enhanced Interactive Python. Type '?' for help.
In [1]:
```

3.2. Check the devices included in the lab's testbed:

```
In [1]: testbed.devices
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>})
```

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

```
In [2]: nx = testbed.devices['nx-osv-1']
In [3]: asa = testbed.devices['asav-1']
```

3.4. Instruct ipyATS to connect to devices:

[9] iPython wrapper to pyATS and Genie:
https://pypi.org/project/ipyats/

```
In [4]: nx.connect()
In [5]: asa.connect()
```

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

```
In [6]: nx.execute('show inventory')
In [7]: asa.execute('show inventory')
```

3.6. Verify the collected information in the output of each command.
Pay attention to the output of both execute methods to be 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'
```

3.7. Use the following command to exit ipyATS shell:

```
In [1]: exit
```

Task 4: Collect show commands from the network devices

In this task we will use the knowledge we have received from the previous task and write a script that collects basic 'show' commands from each device in the testbed.
Output of these commands would be saved in the directory created by the script: /gathered_commands.
These outputs could be used later if you need to compare later 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 testbed.devices object, iterate over all the devices contained in this object to collect an output of the required commands from each device.

4.1. Let's connect to ipyATS and check parts of the code before running the final script.
At the beginning, we would check the structure of testbed.devices object:

```
$ ipyats --testbed pyats_testbed.yaml
In [1]: print(testbed.devices)
```

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

4.2. As you can see from the output in the previous step, 'Device <device_name>' objects are contained as ditionary 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 for loop would be used:

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

To see how the code from the next steps would be input to Python interpreter, copy and paste to ipyats.

Before running it[10], see Python indentation. Example of 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
    ...:
```

Figure 7 – How to check a Python code before running it via pyats

Paste the code shown below into the pyats console:

```
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############################')
```

Study the output to understand the structure of TopologyDict better.
Output for each device is shown inside '############################' stanzas.
Pay attention to the parts marked yellow, 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'>
############################
```

[10] Before pressing the 'Enter' button after the blank line, which instructs ipyats to run a code.

```
#########################
#####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'>
#########################
```

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

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

```
device.connect()
print(device.execute('show inventory'))
```

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 marked yellow. Paste it into the pyats console:

```
In [3]:
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:')
device.execute('show inventory')
print('#########################\n')
```

4.4. Check the result of this code. Now each device should return the output of 'show inventory' command.
Pay attention to the message stating there is a connection to the device already. This is because we have connected to it during the previous step. This connection is left open until ipyats session is finished[11]:

```
#########################
#####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
```

[11] In case of ipyats, until pyats console is closed.

```
Name: "Chassis", DESCR: "ASAv Adaptive Security Virtual Appliance"
PID: ASAv              , VID: V01     , SN: 9AP535X8NA1
asav-1#
#########################
<..>
```

4.5. For the script collecting a large number of commands, it would be preferred to prune the output of the commands to the console.
To achieve it, we would import a standard Python logging module and use device.connectionmgr.log.setLevel(<logging_level>) method to output only errors to the console (logging level "ERROR" would be used for this).

By default level "INFO" is used, that's why all output is sent to the console[12].

We will modify the code in the following way:

```
import logging <- Importing standard logging module

device.connectionmgr.log.setLevel(logging.ERROR) <- set logging level to
ERROR
```

You can check the current (default) logging level in ipyats:

In [1]: import logging

In [2]: testbed.devices['asav-1'].connectionmgr.log.getEffectiveLevel()

Out[2]: 20

Numeric value of 20 corresponds to INFO logging level, see:

https://docs.python.org/3/library/logging.html#logging-levels

After logging level is changed to 'ERROR', it would be reflected in the output as well:

In [3]: testbed.devices['asav-1'].connectionmgr.log.setLevel(logging.ERROR)

In [4]: testbed.devices['asav-1'].connectionmgr.log.getEffectiveLevel()

Out[2]: 40

Numeric value of 20 corresponds to ERROR logging level.

The final code would look as shown below. The added code is marked yellow. Let's paste it into ipyats and see the result:

```
import logging
for device_name, device in testbed.devices.items():
    print('##########################')
print(f'#####device_name = {device_name}, device =
{type(device)}\n#####')
```

[12] See Logging Levels chapter on doc.python.org:
https://docs.python.org/3/library/logging.html#logging-levels

```
device.connect()
print('#####Output:')
device.connectionmgr.log.setLevel(logging.ERROR)
device.execute('show inventory')
print('##########################\n')
```

Now the result looks as it was intended to, no output of the commands shown to the console:

```
##########################
#####device_name = asav-1, device = <class
'genie.libs.conf.device.Device'>
#####
#####Output:
##########################

##########################
#####device_name = csr1000v-1, device = <class
'genie.libs.conf.device.iosxe.device.Device'>
#####
#####Output:
##########################

##########################
#####device_name = nx-osv-1, device = <class
'genie.libs.conf.device.nxos.device.Device'>
#####
#####Output:
##########################
```

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

- key: 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.

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

After the connection to each device has been established, script will check if this device type is specified in commands_to_gather dictionary. If so, it will collect all the commands for this device type. Pay attention we will use INFO logging level, so that we can check that all the commands are gathered. All the other additions to the code from the previous step are marked yellow. Copy and paste this code to the ipyats console. Ensure the result contains the output of both commands from the list for each device type.

```
import logging

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:')
device.connectionmgr.log.setLevel(logging.INFO)
if commands_to_gather.get(device_os):
      for command in commands_to_gather[device_os]:
      device.execute(command)
print('##########################\n')
```

4.7. Now we are ready to study the final version of the script gathering commands specified from all the devices in the testbed and saving them to files on Linux.

Main logic of the script is based on the steps shown above.

Open a prepared script step0_pyats_example.py.

```
$ more step0_pyats_example.py
```

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

The script step0_pyats_example.py has the following Python functions:

def main() – contains commands_to_gather dictionary, where the list of commands for each device type is stored. Calls collect_device_commands function (see "1" on the picture below).

def create_non_existing_dir(dir_path) – this is a supplementary function and it's used to check whether the directory already exists, and if it does not exist, the function tries to create it. In case when it can't be created, the script will throw an error (see "4" on the picture below).

def collect_device_commands(commands_to_gather, dir_name) – the most interesting function for us (see "3" on the picture below).
Main tasks this function performs:

- Creates directory for the gathered commands (calls create_non_existing_dir function for it – see "4" on the picture below)
- Iterates over devices - see "5" (in the way is 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" on the picture below).

For simplicity of the script, name of testbed is hard-coded into the main():
testbed_filename = './pyats_testbed.yaml'

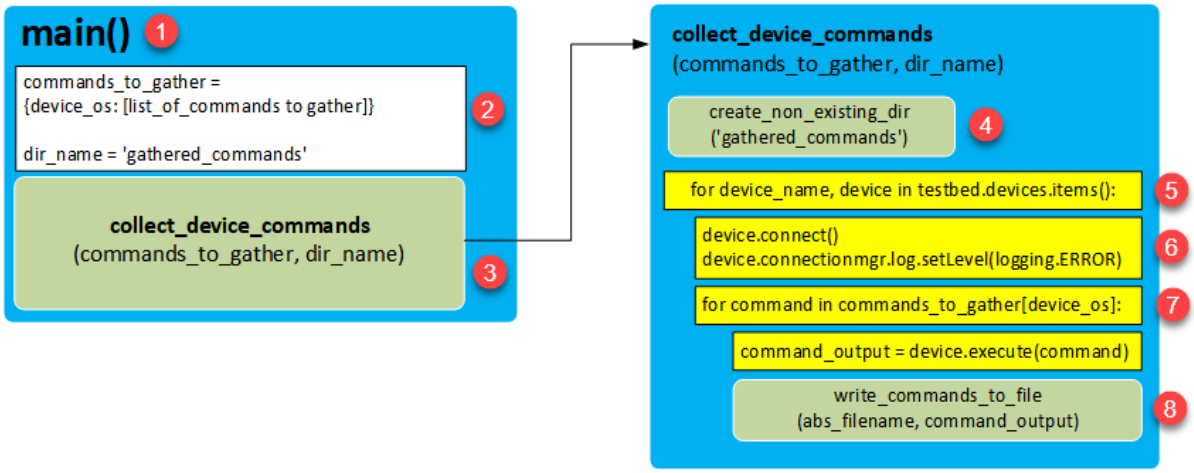


Figure 8 – High-level script "step0_pyats_example.py" structure

4.8. Now run the script:

```
$ python3 step0_pyats_example.py
```

Check there is a new directory created: gathered_commands.

```
ls -l | grep gathered_commands
```

Check time when it was created:

```
(pyats) cisco@win10:~/labpyats$ ls -l | grep gathered_commands
drwxrwxrwx 1 cisco cisco   512 Jan 23 12:08 gathered_commands
```

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

```
ls -l ./gathered_commands/*
```

```
(pyats) cisco@win10:~/labpyats$ ls -l ./gathered_commands/*
./gathered_commands/asav-1:
total 12
-rwxrwx--- 1 cisco cisco  123 Jan 23 12:08 'asav-1_show inventory'
-rwxrwx--- 1 cisco cisco  428 Jan 23 12:08 'asav-1_show license all'
-rwxrwx--- 1 cisco cisco  226 Jan 23 12:08 'asav-1_show ospf neighbor'
-rwxrwx--- 1 cisco cisco 1199 Jan 23 12:08 'asav-1_show route'
-rwxrwx--- 1 cisco cisco 7863 Jan 23 12:08 'asav-1_show running-config'

./gathered_commands/csr1000v-1:
total 16
-rwxrwx--- 1 cisco cisco  334 Jan 23 12:08 'csr1000v-1_show inventory'
-rwxrwx--- 1 cisco cisco  245 Jan 23 12:08 'csr1000v-1_show ip ospf neighbor'
-rwxrwx--- 1 cisco cisco 2614 Jan 23 12:08 'csr1000v-1_show ip route vrf *'
-rwxrwx--- 1 cisco cisco 7384 Jan 23 12:08 'csr1000v-1_show license feature'
-rwxrwx--- 1 cisco cisco 2776 Jan 23 12:08 'csr1000v-1_show running-config'

./gathered_commands/nx-osv-1:
total 24
-rwxrwx--- 1 cisco cisco  651 Jan 23 12:08 'nx-osv-1_show inventory'
-rwxrwx--- 1 cisco cisco  558 Jan 23 12:08 'nx-osv-1_show ip ospf neighbor vrf all'
-rwxrwx--- 1 cisco cisco 2647 Jan 23 12:08 'nx-osv-1_show ip route vrf all'
-rwxrwx--- 1 cisco cisco 1387 Jan 23 12:08 'nx-osv-1_show license usage'
-rwxrwx--- 1 cisco cisco 6556 Jan 23 12:08 'nx-osv-1_show running-config'
```

Figure 9 – Files with output of commands gathered by "step0_pyats_example.py" script

Task 5: Write first test script using pyATS library

Now let's write our first test script on Python using pyATS library. Our test script will connect to all the devices in testbed and print 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 pyATS test script file.
pyATS test script is a file with the Python code which uses pyATS library.
Structure of pyATS test script is modular and straightforward.

Each testscript is written in a Python file and split into three major sections (Python classes)[13] – see Figure 10 for graphical representation:

- Common Setup (the first section in the test script, run at the beginning, performs all the "common" setups required for the script).
- Testcase(s) – a self-contained individual unit of testing each testcase is independent from 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 class).

Both Common Setup and Common Cleanup could be only one in a script, whereas it might be multiple Testcases in one test script.

[13] Sections are executed in the sequence shown below.

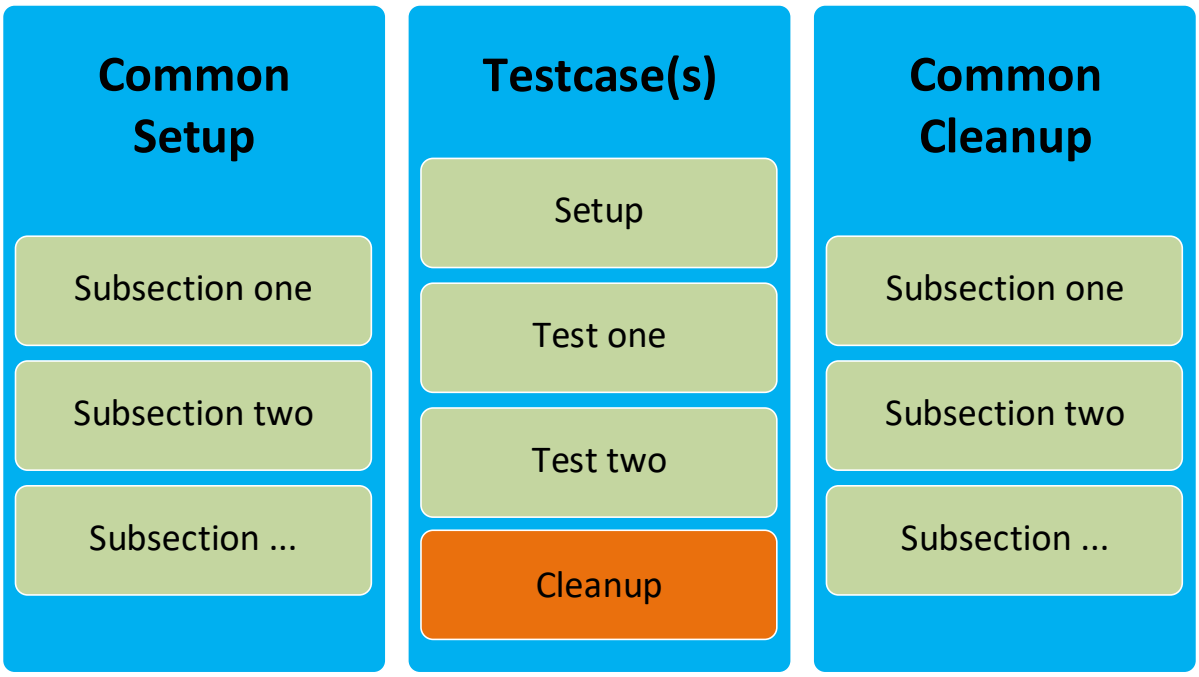


Figure 10 – pyATS testscript high-level structure

5.1 Let's verify our first testscript. Open file empty_pyats_example.py and observe its structure:

```
$ more empty_pyats_example.py
```

5.2 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 MyCommonSetup which is inherited from aetest.CommonSetup represents the major section "Common Setup" (see Figure 10).
Python class MyCommonSetup is where initializations and preparations before the actual script's testcases should be performed. For this reason MyCommonSetup is always run first, before all the testcases.
Refer to the description of the code of this Python class shown below:

```
class MyCommonSetup(aetest.CommonSetup):
    @aetest.subsection
    def establish_connections(self,testbed):
        # Load testbed file which is passed as commmand-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(
                "Connect to device '{d}'".format(d=device.name)))
            try:
                device.connect()
            except Exception as e:
                self.failed("Failed to establish connection to
'{}'".format(
                    device.name))
            device_list.append(device)
        # Pass list of devices to testcases
        self.parent.parameters.update(dev=device_list)
```

5.3 Let's run our first testscript! This testscript will try to connect to all the devices in the testbed and print the results of these attempts:

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

Upon finishing the testscript, pyATS generates a report of Success/Failed testcases, MyCommonSetup 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 Figure 11.

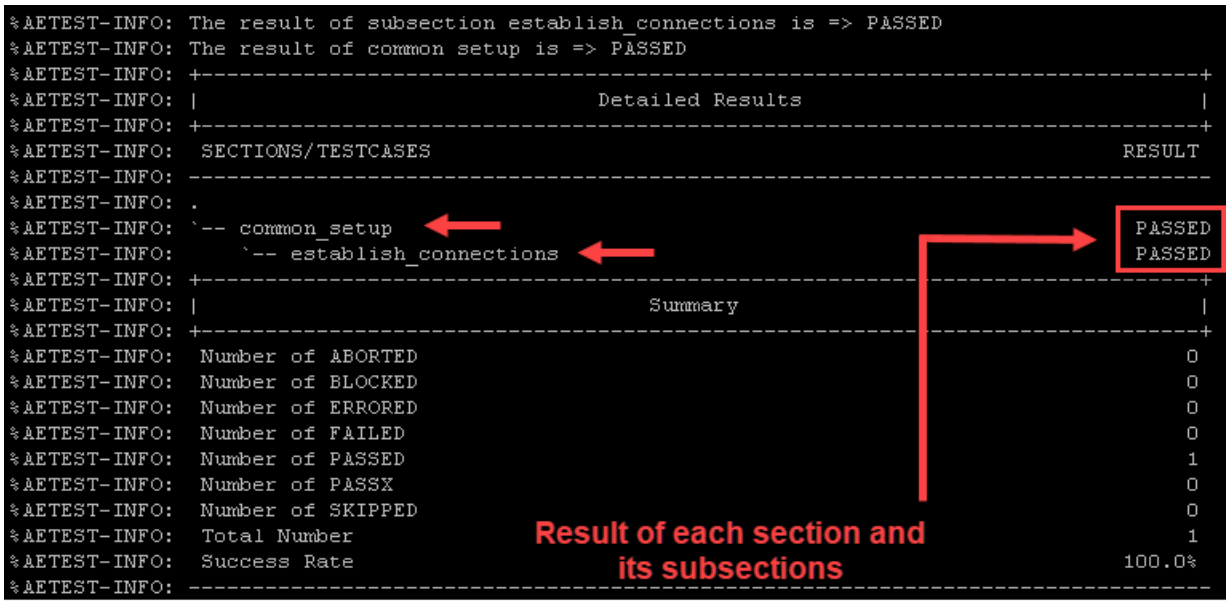


Figure 11 – Result of the testscript's "empty_pyats_example.py" running

Task 6: Verify log messages

In this testcase we will need to verify that the logging messages with ERROR or WARN are not present on the devices in the testbed.
High-level logic of the test 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, else the test should succeed.

6.1 Prior to creating our testcase, connect to ASA, using virlutils and clear logs on it:

```
$ virl console asav-1
Attempting to connect to console of asav-1
Trying 198.18.134.1...
Connected to virl.demo.dcloud.cisco.com.
Escape character is '^]'.

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

In order to quit the console of a device while connected using '**virl console**', press **CTRL** and **]** buttons simultaneously, then type quit and console would be disconnected.

See example below:

```
asav-1# <- CTRL and ] buttons pressed simultaneously here
telnet> quit
Connection closed.
```

6.2 Let's connect to ipyATS and check our idea:

```
$ ipyats --testbed pyats_testbed.yaml
In [1]: csr = testbed.devices['csr1000v-1']
In [2]: asa = testbed.devices['asav-1']
In [3]: csr.connect()
In [4]: asa.connect()
```

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

```
In [5]: out1 = csr.execute('show logging | i ERROR|WARN')
In [6]: out2 = asa.execute('show logging | i ERROR|WARN')
```

Output for ASA should be empty.

If you don't see any ERROR logs on CSR1000v-1 device, then connect to it and generate test ERROR message:

$ virl console csr1000v-1

csr1000v-1# send log 'Test ERROR message for PyATS'

Then repeat previous step in **ipyATS:**

In [7]: out1 = csr.execute('show logging | i ERROR|WARN')

To check whether there is an empty or non-empty output, we will use Python len() built-in function, which returns the length of the given string.

If collected output is empty, then **len()** of the output will be 0, otherwise the result will be greater than 0:

```
In [7]: len(out2)
Out [7]: 0
In [8]: len(out1)
Out [8]: 664
```

6.4 Use the following command to exit the ipyats shell:

```
In [9]: exit
```

6.5 Open file step1_pyats_example.py in Vim editor

```
$ vim step1_pyats_example.py
```

This file reuses establish_connections(self,testbed) method from empty_pyats_example.py, which make connections to all the devices in the testbed. Also it has an initial version of the code for class VerifyLogging

Pay attention to the method **self.parent.parameters.update(dev=device_list)**, located at the end of **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 could be passed between different classes.

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

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

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

6.6 Let's add content to class VerifyLogging, to implement approach, that we tested on ipyATS. Use arrows ↑→↓← on the keyboard to find the following position in the code (marked yellow below) and then press "i" button on the keyboard, to instruct Vim to enter "insert mode" and start adding text where the cursor is:

```
class VerifyLogging(aetest.Testcase):

    @aetest.test
    def error_logs(self):
        <<Insert next snippet here. Look out for indentations!>>
```

6.7 Insert the next snippet into the found position:

```
        any_device = self.parent.parameters['dev'][0]
        output = any_device.execute('show logging | i ERROR|WARN')

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

6.8 Save changes in file step1_pyats_example.py:
press [Esc] button on your keyboard, then type:

```
:wq
```

6.9 Execute the created testscript and check the results section:

```
$ python3 step1_pyats_example.py --testbed pyats_testbed.yaml
```

Testcase error_log will run only for one device, scroll above the result's section and you will see to which device this output is related:

```
[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 testcase for only one device, now we need to get familiar with aetest.loop method, which will let us repeat an elementary testcase (written for one device) for every device in the testbed. Sounds promising, yeah?!

6.10 Open file step1_pyats_example.py once again:

```
$ vim step1_pyats_example.py
```

And modify error_logs method, to receive device argument on input and collect commands for this device:

Press **"i"** button on keyboard, to instruct Vim to start inserting the text where the cursor is.

```
    @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
```

6.11 Next, we need to modify setup(self) method of class VerifyLogging, to load all the devices from the testbed and run error_logs method for each device.

Do not forget to delete **pass** statement from the body of **def setup(self):**

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

6.12 Save changes in file step1_pyats_example.py:
press [Esc] button on your keyboard, then type:

```
:wq
```

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

6.13 Execute created testscript, testcase error_log will run for all the devices in the testbed:

```
$ python3 step1_pyats_example.py --testbed pyats_testbed.yaml
```

6.14 Check VerifyLogging results' section, 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$
```

Figure 12 – VerifyLogging testcase results

Task 7: Verify the service contracts coverage

In this testcase we have the list of devices' serial numbers, covered by the service contracts, and we need to verify, that all the devices in the testbed are covered by the service contracts. This ensures you to 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 outputs of *'show inventory'* to find the device's serial number (SN).
- Verify whether SN is in the list, covered by the service contracts.

7.1 Let's connect to ipyATS and check our idea:

```
$ ipyats --testbed pyats_testbed.yaml
In [1]: csr = testbed.devices['csr1000v-1']
In [2]: asa = testbed.devices['asav-1']
In [3]: nx = testbed.devices['nx-osv-1']
In [4]: csr.connect()
In [5]: asa.connect()
In [6]: nx.connect()
```

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

```
In [7]: out1 = csr.parse('show inventory')
In [8]: out2 = asa.parse('show inventory')
In [9]: out3 = nx.parse('show inventory')
```

7.3 Now we can observe the structure of parsed outputs.

Python library **pprint** would be imported in this task.

This is used to break the output (Python dictionary) onto multiple lines, which is more easy to check, comparing to having it in one line.

We are starting with the parsed output for CSR1000V. Review it and pay attention to the parts marked yellow.

```
In [10]: import pprint
In [11]: pprint.pprint(out1)
{'main': {'chassis': {'CSR1000V': {'descr': 'Cisco CSR1000V Chassis',
                                   'name': 'Chassis',
                                   'pid': 'CSR1000V',
                                   'sn': '9KZZ4X737UP',
                                   'vid': 'V00'}}},
In [12]: out1['main']['chassis']['CSR1000V']['sn']
'9KZZ4X737UP'
```

Review the parsed output for ASAv:

```
In [13]: pprint.pprint(out2)
{'Chassis': {'description': 'ASAv Adaptive Security Virtual Appliance',
             'pid': 'ASAv',
             'sn': '9ARUJ17VVFS',
             'vid': 'V01'}}
In [14]: out2['Chassis']['sn']
'9ARUJ17VVFS'
```

Review the parsed parsed output for NX-OSv:

```
In [15]: pprint.pprint(out3)
{'name': {'Chassis': {'description': 'Nexus9000 9000v Chassis',
                      'pid': 'N9K-9000v',
                      'serial_number': '9712TV4C2JF',
                      'slot': 'None',
                      'vid': 'V02'},
In [16]: out3['name']['Chassis']['serial_number']
'9712TV4C2JF'
```

Now we have all the needed information to write the next testscript, let's get the ball rolling!

7.4 Use the following command to exit ipyATS shell:

```
In [9]: exit
```

7.5 Open file step2_pyats_example.py in Vim editor:

```
$ vim step2_pyats_example.py
```

7.6 Review the content of the inventory testcase, note that we use the data structure learned from ipyATS in the previos step, to extract a serial number from the output of *show inventory*:

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

```
    <…>
        out1 = device.parse('show inventory')
        chassis_sn = out1['main']['chassis']['CSR1000V']['sn']
```

Complete this testcase by replacing <<replace me>> instructions in the code for 'nxos' and 'asa' devices. To accomplish this, you need to copy path to the serial number from the structures, explored in the previous step with ipyATS (out2 and out3 variables).

7.7 When you finish, save the changes to the file: step2_pyats_example.py

Press [Esc] button on your keyboard to exit Vim “insert mode”, then type:

```
:wq
```

7.8 Execute the created testscript and check “Detailed Results” section:

```
$ python3 step2_pyats_example.py --testbed pyats_testbed.yaml
```

What are the results of these testcases? All fails? Do you have a clue, why?
(Scroll below for the right answer).

All the tests have failed, since we have SNs from a different network, in our contract SNs list at the beginning of step2_pyats_example.py file:

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

7.9 Replace the serial numbers in list contract_sn with SNs from our testbed's equipment and execute the changed testscript once again:

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 step2_pyats_example.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$
```

Figure 13 – Example of all testcases from "step2_pyats_example.py" passed

Task 8: Verify the routing information using parsers and Genie learn

In this testcase we have the list of critical routes (usually this is a device's loopback interface) and we need to 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 device's RIB.

8.1. Let's connect to ipyATS and check our idea:

```
$ ipyats --testbed pyats_testbed.yaml
In [1]: csr = testbed.devices['csr1000v-1']
In [2]: asa = testbed.devices['asav-1']
In [3]: nx = testbed.devices['nx-osv-1']
In [4]: csr.connect()
In [5]: asa.connect()
In [6]: nx.connect()
```

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

```
In [7]: csr_routes = csr.learn('routing')
In [8]: nx_routes = nx.learn('routing')
```

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

```
In [9]: import pprint
In [10]: pprint.pprint(csr_routes.info)
Out [10]:
{'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 [11]: 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 [12]: pprint.pprint
(nx_routes.info['vrf']['default']['address_family']['ipv4']['routes'])
```

8.3. Use the following command to exit the ipyats shell:

```
In [13]: exit
```

8.4. Open file step3_pyats_example.py in Vim editor

```
$ vim step3_pyats_example.py
```

8.5. 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 get the snapshot of routing feature:

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

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

         output = device.learn('routing')
         rib = <<replace me>>
```

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 testcase 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
```

8.6. Complete this testcase by replacing <<replace me>> sign for rib variable. To accomplish this, you need to copy the path to the rib routes, explored in the previous step (output.info['vrf']['default']['address_family']['ipv4']['routes']).

Press **"i"** button on keyboard, to instruct Vim to start inserting of text where the cursor is.

8.7. When you finish, save changes to file step3_pyats_example.py and execute the created testscript and check the results section:

press [Esc] button on your keyboard, then type:

```
:wq
$ python3 step3_pyats_example.py --testbed pyats_testbed.yaml
```

Task 9: Run ping to verify reachability

In this testcase we need to 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.

9.1. Let's connect to ipyATS and check our idea:

```
$ ipyats --testbed pyats_testbed.yaml
In [1]: csr = testbed.devices['csr1000v-1']
In [2]: nx = testbed.devices['nx-osv-1']
```

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

```
In [3]: nx.find_links(csr)
{<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>}
```

9.2. Use the following command to exit ipyATS shell:

```
In [4]: exit
```

Before studying the code and running the next script, let's dive into the details how information about topology is stored in testbed file (see Figure 14 for the graphical representation of the explanation following below):

Things to know about structure of the testbed file (testbed.yaml):

- PyATS Testbed object contains a Python dictionary devices[14].
- Elements of the devices dictionary are the Device objects.
- Each object in the devices stores dictionary of the interfaces objects.
- Each interfaces object stores the link object.

[14] Called using testbed.devices from ipyats shell.

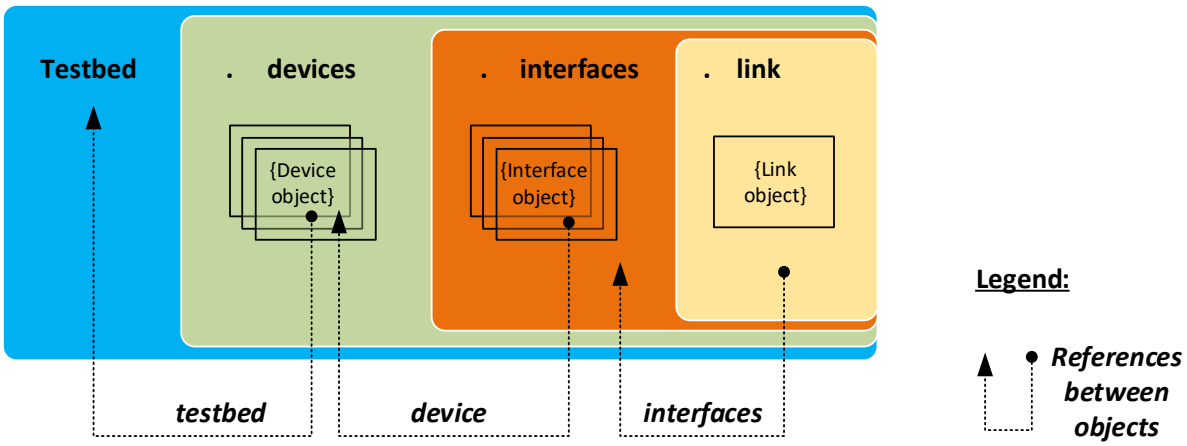


Figure 14 – Interactions between objects in pyATS Testbed

- 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. Eg: Ethernet, ATM, Loopback.
- Link objects represent the connection (wire) between two or more interfaces within a testbed topology.

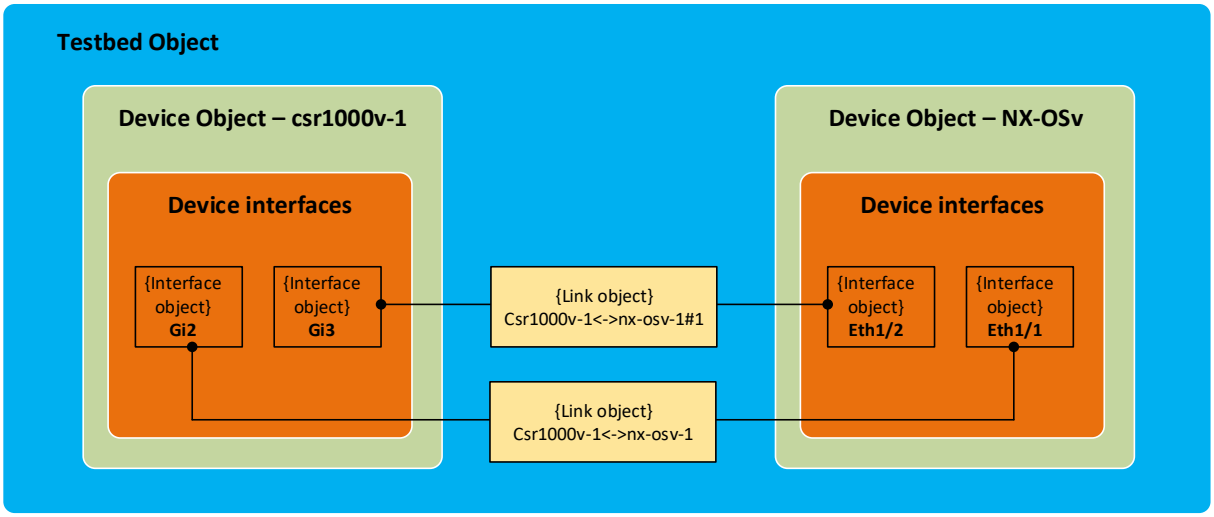


Figure 15 – pyATS objects and topology

Note one important fact: we can get value of an attribute of each object, for example, we can get a link to which an interface object belongs by calling interface.link attribute. And also we can reference interfaces which belong to this link, by calling link.interfaces reference.

Open file step4_pyats_example.py using Linux more utility:

```
$ more step4_pyats_example.py
```

9.3. Review the content of PingTestcase testcase, take a look on def setup(self) function, where we get all the links between NX-OSv and CSR1000V.
Then we 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.

```
        # Find links between Nx-os device and CSR100v
        dest_links = nx.find_links(csr)
        dest_ips = []

        for links in dest_links:
           # process each link between devices

           for iface in links.interfaces:
              # process each interface (side) of the linki
              if iface.ipv4 is not None:
                print(f'{iface.name}:{iface.ipv4.ip}')
                dest_ips.append(iface.ipv4.ip)
              else:
                print(f'Skipping iface {iface.name} without IPv4 address')
```

In function def ping(self,dest_ip) we execute ping command for each IPv4 address of both ends of the links between NX-OSv and CSR1000V. The string returned by ping is shown below. The field that has to be extracted is marked yellow:

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

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

```
        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}%')
```

9.4. Execute the created testscript and check the results section, all pings should succeed:

```
$ python3 step4_pyats_example.py --testbed pyats_testbed.yaml
```

Summary

During this lab you have gotten hands-on experience with PyATS and Genie - vendor-agnostic suite of libraries for Python.

You have learned how to build your own automated tests, using PyATS/Genie frameworks. These tools open wide opportunities ("sky is the limit"), and it's not hard to start using them.
This lab has introduced you to real-world examples and, we hope, has given you fast start for automation of tests in your network.

The main things we wanted to emphasize in this lab:

- Test automation for network operations is available today.
- It's easy to implement automation in your network, with little to no programming experience.
- PyATS/Genie are simple and extensible frameworks to start automation with.

In case you need to check the content of this lab after Cisco Live, it's available on GitHub (see the following chapter: "Links to the LABOPS-1775 Project on GitHub").
You will get access to the lab source code from there and would be able to use it in your network afterwards.

We hope this information was useful and you will use it in your job.

Happy programming and good luck with learning and using PyATS/Genie!

Please, don't forget to rate the lab so that we can make it better.

Links to the LABOPS-1775 Project on GitHub

URL to Clone or download using git client: https://github.com/cleur2293/labpyats.git
URL to open in a browser: https://github.com/cleur2293/labpyats

Up to date version of the code and lab guide are always available using these links. Feel free to use it in your Projects.

Related Sessions at Ciscolive

- Network Assurance: pyATS/Genie for Network Engineers
  DEVNET-1204.
- DevNet Workshop: NetDevOps programming with pyATS/Genie for beginners
  DEVWKS-2808.
- DevNet Workshop: Network Verification with pyATS/Genie for non-programmers
  DEVWKS-2595.
- DevNet Workshop: pyATS Intro - creating parsers with GENIE
  DEVWKS-2601.
- DevNet Workshop: Utilizing Cisco CXTA service framework to validate network elements
  DEVWKS-1407.