Automating Testing

7.6.1

Automated Test and Validation

In this topic, you will learn about a suite of products and practices created by Cisco and its user community to extend test automation to software-driven network configuration, and to reduce or eliminate uncertainty about how prospective network architectures will function and perform when fully implemented.

Testing infrastructure challenges

Automation tools like Ansible, Puppet, Chef, and others solve part of the problem by turning infrastructure into code. But DevOps typically needs more fine-grained ways to define and implement infrastructures, certify that deployed infrastructures are working as required, and proactively ensure its smooth operations. DevOps also needs ways to preemptively take action when failures are imminent, and find and fix issues when errors occur.

When you use unit-testing tools like pytest in tandem with higher-order automation and in concert with continuous delivery (CI/CD), you can build environments where code can be automatically tested when changes are made.

Unit-testing frameworks make tests a part of your codebase, following the code through developer commits, pull requests, and code-review gates to QA/test and Production. This is especially useful in test-driven development (TDD) environments, where writing tests is a continuous process that actually *leads* development, automatically encouraging very high levels of test coverage.

The challenges of testing a network

The behavior and performance of a real-world network is collective, maintained by the configurations of many discrete pieces of equipment and software.

In traditional environments, connectivity and functionality are manually maintained across numerous individual pieces of equipment via diverse interfaces. Often, operations require that changes be made on in-service hardware carrying live traffic. This is difficult, time-consuming, extremely error-prone, and risky.

Network misconfigurations are often only discovered indirectly, when computers cannot talk to one another. As networks become more complex and carry more diverse and performance-sensitive traffic,

risks to security and performance degradations, which may be both difficult to discover and to quantify, are increasingly important consequences of misconfiguration.

Network management and testing are still complex even when network devices and connectivity become software-addressable and virtualized. Methods of building and configuring networks certainly change, but you still need to create a collective architecture for safe connectivity by touching numerous device interfaces in detailed ways.

Testing Software Defined Networks (SDN)

Cisco has made huge strides in developing Software Defined Networks (SDN) and middleware that let engineers address a physical network collective as a single programmable entity. In Cisco's case, this includes:

- **Application Centric Infrastructure (ACI)** – This is a comprehensive data center solution that runs on top of Nexus 9000 and APIC-enabled devices. It enables abstraction and programmability of total network functionality via the Application Policy Infrastructure Controller (APIC).
- **Digital Network Architecture Center (Cisco DNA Center)** – This is an open, extensible, software driven architecture for Catalyst 9000 and other IOSXE devices for enterprise network.
- REST API and SDKs enabling integration with automation tools like Ansible, Puppet, and Chef.

Solutions like ACI manage the whole network by converging models (often written in a declarative syntax called YANG) that represent desired states of functionality and connectivity. The middleware enables a model to work harmoniously with other models that are currently defining system state and resource requirements. Engineers interact less often with individual devices directly, though the models still need to provide enough detail to enable configuration. The complex, fast-evolving state of large infrastructures can be maintained as code, enabling:

- Rapid re-convergence to desired states at need. If a device breaks and is replaced, it can be rapidly reintegrated with an existing network and its programmed functionality quickly restored, along with network behavior and performance.
- Portability, so that when a core application moves from one data center to another, its required network configuration accompanies it.
- Version control, CI/CD, and other tools to maintain, evolve, and apply the network codebase.

These innovations are increasingly popular with larger organizations, carriers, and other scaled-up entities. In many cases, however, networks still comprise several generations of diverse, hybrid, multi-vendor physical and virtual infrastructures, so the problem of deliberate, device-level configuration still looms.

And even when sophisticated SDN is available, SDN controller/orchestrators cannot always prevent misconfiguration. They can reject flawed code, perform sanity checks before applying changes, and inform when models make demands exceeding resource thresholds, but seemingly legitimate changes can still be applied.

A network test solution: pyATS

Python Automated Test System (pyATS) is a Python-based network device test and validation solution, originally developed by Cisco for internal use, then made available to the public and partially open-sourced. pyATS can be used to help check whether your changes work before putting them into production, and continue validation and monitoring in production to ensure smooth operations.

The pyATS ecology

pyATS originated as the low-level Python underpinning for the test system as a whole. Its higher-level library system, Genie, provides the necessary APIs and libraries that drive and interact with network devices, and perform the actual testing. The two together form the Cisco test solution we know as pyATS.

pyATS has several key features:

- pyATS framework and libraries can be leveraged within any Python code.
- It is modular, and includes components such as:
  - AEtest executes the test scripts.
  - Easypy is the runtime engine that enables parallel execution of multiple scripts, collects logs in one place, and provides a central point from which to inject changes to the topology under test.

- A CLI enables rapid interrogation of live networks, extraction of facts, and helps automate running of test scripts and other forensics. This enables very rapid 'no-code' debugging and correction of issues in network topologies created and maintained using these tools.

In SDN/cloud/virtual network environments, setup can involve actually building a topology, and cleanup can involve retiring it, reclaiming platform resources. This setup and cleanup can be done directly using pyATS code. pyATS provides an enormous interface library to Cisco and other infrastructure via a range of interfaces, including low-level CLI and REST APIs, as well as connectors to ACI and other higher-order SDN management frameworks.

pyATS can consume, parse, and implement topologies described in JSON, as YANG models, and from other sources, even from Excel spreadsheets.

pyATS can also be integrated with automation tools like Ansible for building, provisioning, and teardown. However, it may be better practice to do the reverse. Use Ansible, Puppet, or Chef to manage the infrastructure's entire codebase and have those products invoke Python (and pyATS) to deal with the details of network implementation. These tools also recruit ACI or other middleware to simplify the task, and permit segregated storage and versioning of YANG or other models defining concrete topologies.

Alternatively, you can invoke pyATS indirectly in several ways (including ways requiring minimal Python programming knowledge).

7.6.2

pyATS Example

The following content shows how to use pyATS to create and apply tests. You will need to be familiar with this information to help you complete the lab on the next page. Simply read along with this example to better understand pyATS.

Virtual environments

The pyATS tool is best installed for personal work inside a Python virtual environment (venv). A venv is an environment copied from your base environment, but kept separate from it. This enables you to avoid installing software that might permanently change the state of your system. Virtual environments exist in folders in your file system. When they are created, they can be activated, configured at will, and components installed in them can be updated or modified without changes being reflected in your host's configuration. The ability to create virtual environments is native to Python 3, but Ubuntu 18.04 may require you to install a `python3-venv` package separately.

The following instructions describe how to create a venv on Ubuntu 18.04 (where `python3` is the default command). If you are using a different operating system, refer to the appropriate documentation for pip and virtual environments.

Ensure that `python3-pip`, the Python3 package manager, is in place. You would also install `git`, which you would need later:

```
sudo apt-get install python3-pip
sudo apt-get install python3-venv
sudo apt-get install git
```

Create a new virtual environment in the directory of your choice. In this example, it is called `myproject`.

```
python3 -m venv myproject
```

Venv creates the specified working directory (`myproject`) and corresponding folder structure containing convenience functions and artifacts describing this environment's configuration. At this point, you can cd to the `myproject` folder and activate the venv:

```
cd myproject
source bin/activate
```

Installing pyATS

You can install pyATS from the public Pip package repository (PyPI).

Note: You may see "Failed building wheel for ... `<wheelname>"` errors while installing pyATS through pip. You can safely ignore those errors as `pip` has a backup plan for those failures and the dependencies are installed despite errors reported.

```
pip install pyats[full]
```

Verify that it was installed by listing the help:

```
pyats --help
```

Clone the pyATS sample scripts repo, maintained by Cisco DevNet, which contains sample files you can examine:

```
git clone https://github.com/CiscoDevNet/pyats-sample-scripts.git
cd pyats-sample-scripts
```

The installed target, `pyats[full]`, includes both the low-level underpinnings, various components, dependencies, and the high-level Genie libraries.

pyATS test case syntax

The test declaration syntax for pyATS is inspired by that of popular Python unit-testing frameworks like `pytest`. It supports basic testing statements, such as an assertion that a variable has a given value, and adds to that the ability to explicitly provide results (including result reason, and data) via specific APIs. This is demonstrated in the following excerpt from a basic test script. The pyATS test script can be found in `/basics/pyats-sample-script.py` from the repository that you cloned previously. A portion of the script is shown below.

```
class MyTestcase(aetest.Testcase):
    @aetest.setup
    def setup(self, section):
        '''setup section
        create a setup section by defining a method and decorating it with
        @aetest.setup decorator. The method should be named 'setup' as good
        convention.
        setup sections are optional within a testcase, and is always runs first.
        '''
        log.info("%s testcase setup/preparation" % self.uid)
        # set some variables
        self.a = 1
        self.b = 2
    @aetest.test
    def test_1(self, section):
        '''test section
        create a test section by defining a method and decorating it with
        @aetest.test decorator. The name of the method becomes the unique id
        labelling this test. There may be arbitrary number of tests within a
        testcase.
        test sections run in the order they appear within a testcase body.
        '''
        log.info("test section: %s in testcase %s" % (section.uid, self.uid))
        # testcase instance is preserved, eg
        assert self.a == 1
    @aetest.test
    def test_2(self, section):
        '''
        you can also provide explicit results, reason and data using result API.
        These information will be captured in the result summary.
        '''
        log.info("test section: %s in testcase %s" % (section.uid, self.uid))
        if self.b == 2:
            self.passed('variable b contains the expected value',
                        data = {'b': self.b})
```

```
        else:
            self.failed('variable b did not contains the expected value',
                        data = {'b': self.b})
```

If you click through and examine the entire test script, you will see it contains several sections:

- A common Setup block
- Multiple Testing blocks
- A common Cleanup block

These blocks contain statements that prepare and/or determine readiness of the test topology (a process that can include problem injection), perform tests, and then return the topology to a known state.

The Testing blocks, which are often referred to in pyATS documentation as the Test Cases, can each contain multiple tests, with their own Setup and Cleanup code. Best practice suggests that the common Cleanup section, at the end, be designed for idempotency. This means that it should check and restore all changes made by Setup and Test, restoring the topology to its original, desired state.

pyATS scripts and jobs

A pyATS script is a Python file where pyATS tests are declared. It can be run directly as a standalone Python script file, generating output only to your terminal window. Alternatively, one or more pyATS scripts can be compiled into a "job" and run together as a batch, through the pyATS EasyPy module. This enables parallel execution of multiple scripts, collects logs in one place, and provides a central point from which to inject changes to the topology under test.

The pyATS job file can be found in `/basics/pyats-sample-job.py` from the repository that you cloned previously. A portion of the job file is shown below.

```
import os
from pyats.easypy import run
def main():
    '''
    main() function is the default pyATS job file entry point that Easypy module
consumes
    '''
    # find the location of the script in relation to the job file
    script_path = os.path.dirname(os.path.abspath(__file__))
    testscript = os.path.join(script_path, 'basic_example_script.py')
    # execute the test script
    run(testscript=testscript)
```

If you have performed the installation steps and are now in a virtual environment containing the cloned repo, you can run this job manually to invoke the basic test case:

```
pyats run job basic/basic_example_job.py
```

If you see an error like `RuntimeError: Jobfile 'basic_example_script' did not define main()`, it means you have run the `basic_example_script.py` file rather than the `basic_example_job.py` file. Or, if you see `The provided jobfile 'pyats-sample-scripts/basic/basic_example_job.py' does not exist.` double-check which directory you are working within. Perhaps you have already changed directories into the `pyats-sample-scripts` repository directory.

```
Output
2020-03-01T12:38:50: %EASYPY-INFO: Starting job run: basic_example_job 2020-03-
01T12:38:50: %EASYPY-INFO: Runinfo directory:
/Users/agentle/.pyats/runinfo/basic_example_job.2020Mar01_12:38:48.974991 2020-03-
01T12:38:50: %EASYPY-INFO: ----------------------------------------------------------
----------------------- 2020-03-01T12:38:51: %EASYPY-INFO: Starting task
execution: Task-1 2020-03-01T12:38:51: %EASYPY-INFO:     test harness =
pyats.aetest 2020-03-01T12:38:51: %EASYPY-INFO:     testscript   =
/Users/agentle/src/pyats-sample-scripts/basic/basic_example_script.py 2020-03-
01T12:38:51: %AETEST-INFO: +-------------------------------------------------------
----------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting common setup                                | 2020-03-01T12:38:51: %AETEST-
INFO: +-------------------------------------------------------------------------
--+ 2020-03-01T12:38:51: %AETEST-INFO: +---------------------------------------
----------------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting subsection subsection_1                     | 2020-03-01T12:38:51:
%AETEST-INFO: +-------------------------------------------------------------------
----------+ 2020-03-01T12:38:51: %SCRIPT-INFO: hello world! 2020-03-01T12:38:51:
%AETEST-INFO: The result of subsection subsection_1 is => PASSED 2020-03-
01T12:38:51: %AETEST-INFO: +-------------------------------------------------------
----------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting subsection subsection_2                     | 2020-03-01T12:38:51:
%AETEST-INFO: +-------------------------------------------------------------------
----------+ 2020-03-01T12:38:51: %SCRIPT-INFO: inside subsection subsection_2 2020-
03-01T12:38:51: %AETEST-INFO: The result of subsection subsection_2 is => PASSED
2020-03-01T12:38:51: %AETEST-INFO: The result of common setup is => PASSED 2020-03-
01T12:38:51: %AETEST-INFO: +-------------------------------------------------------
----------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting testcase Testcase_One                       | 2020-03-01T12:38:51:
%AETEST-INFO: +-------------------------------------------------------------------
----------+ 2020-03-01T12:38:51: %AETEST-INFO: +-----------------------------------
----------------------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting section setup                               | 2020-03-01T12:38:51: %AETEST-
INFO: +-------------------------------------------------------------------------
--+ 2020-03-01T12:38:51: %SCRIPT-INFO: Testcase_One testcase setup/preparation
2020-03-01T12:38:51: %AETEST-INFO: The result of section setup is => PASSED 2020-
03-01T12:38:51: %AETEST-INFO: +-------------------------------------------------
-------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting section test_1                              | 2020-03-01T12:38:51: %AETEST-
INFO: +-------------------------------------------------------------------------
--+ 2020-03-01T12:38:51: %SCRIPT-INFO: test section: test_1 in testcase
Testcase_One 2020-03-01T12:38:51: %AETEST-INFO: The result of section test_1 is =>
PASSED 2020-03-01T12:38:51: %AETEST-INFO: +-----------------------------------
----------------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting section test_2                              | 2020-03-01T12:38:51: %AETEST-
```

```
INFO: +------------------------------------------------------------------------
--+ 2020-03-01T12:38:51: %SCRIPT-INFO: test section: test_2 in testcase
Testcase_One 2020-03-01T12:38:51: %AETEST-INFO: Passed reason: variable b contains
the expected value 2020-03-01T12:38:51: %AETEST-INFO: The result of section test_2
is => PASSED 2020-03-01T12:38:51: %AETEST-INFO: +----------------------------------
----------------------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting section cleanup                          | 2020-03-01T12:38:51: %AETEST-
INFO: +------------------------------------------------------------------------
--+ 2020-03-01T12:38:51: %SCRIPT-INFO: Testcase_One testcase cleanup/teardown 2020-
03-01T12:38:51: %AETEST-INFO: The result of section cleanup is => PASSED 2020-03-
01T12:38:51: %AETEST-INFO: The result of testcase Testcase_One is => PASSED 2020-
03-01T12:38:51: %AETEST-INFO: +----------------------------------------------------
-------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting common cleanup                           | 2020-03-01T12:38:51: %AETEST-
INFO: +------------------------------------------------------------------------
--+ 2020-03-01T12:38:51: %AETEST-INFO: +-----------------------------------------
---------------------------------+ 2020-03-01T12:38:51: %AETEST-INFO: |
Starting subsection clean_everything                | 2020-03-01T12:38:51:
%AETEST-INFO: +------------------------------------------------------------------
----------+ 2020-03-01T12:38:51: %SCRIPT-INFO: goodbye world 2020-03-01T12:38:51:
%AETEST-INFO: The result of subsection clean_everything is => PASSED 2020-03-
01T12:38:51: %AETEST-INFO: The result of common cleanup is => PASSED 2020-03-
01T12:38:51: %EASYPY-INFO: ----------------------------------------------------
------------------------ 2020-03-01T12:38:51: %EASYPY-INFO: Job finished. Wrapping
up... 2020-03-01T12:38:52: %EASYPY-INFO: Creating archive file:
/Users/agentle/.pyats/archive/20-
Mar/basic_example_job.2020Mar01_12:38:48.974991.zip 2020-03-01T12:38:52: %EASYPY-
INFO: +------------------------------------------------------------------------
--+ 2020-03-01T12:38:52: %EASYPY-INFO: |                                Easypy
Report                                | 2020-03-01T12:38:52: %EASYPY-INFO: +------
-------------------------------------------------------------------+ 2020-03-
01T12:38:52: %EASYPY-INFO: pyATS Instance   :
/Users/agentle/.local/share/virtualenvs/pyats-sample-scripts-b4vw68FQ/bin/.. 2020-
03-01T12:38:52: %EASYPY-INFO: Python Version   : cpython-3.8.1 (64bit) 2020-03-
01T12:38:52: %EASYPY-INFO: CLI Arguments    :
/Users/agentle/.local/share/virtualenvs/pyats-sample-scripts-b4vw68FQ/bin/pyats run
job basic/basic_example_job.py 2020-03-01T12:38:52: %EASYPY-INFO: User
: agentle 2020-03-01T12:38:52: %EASYPY-INFO: Host Server      : AGENTLE-M-339A
2020-03-01T12:38:52: %EASYPY-INFO: Host OS Version  : Mac OSX 10.14.6 (x86_64)
2020-03-01T12:38:52: %EASYPY-INFO: 2020-03-01T12:38:52: %EASYPY-INFO: Job
Information 2020-03-01T12:38:52: %EASYPY-INFO:     Name         : basic_example_job
2020-03-01T12:38:52: %EASYPY-INFO:     Start time   : 2020-03-01 12:38:50.019013
2020-03-01T12:38:52: %EASYPY-INFO:     Stop time    : 2020-03-01 12:38:51.732162
2020-03-01T12:38:52: %EASYPY-INFO:     Elapsed time : 0:00:01.713149 2020-03-
01T12:38:52: %EASYPY-INFO:     Archive      : /Users/agentle/.pyats/archive/20-
Mar/basic_example_job.2020Mar01_12:38:48.974991.zip 2020-03-01T12:38:52: %EASYPY-
INFO: 2020-03-01T12:38:52: %EASYPY-INFO: Total Tasks    : 1 2020-03-01T12:38:52:
%EASYPY-INFO: 2020-03-01T12:38:52: %EASYPY-INFO: Overall Stats 2020-03-01T12:38:52:
%EASYPY-INFO:     Passed     : 3 2020-03-01T12:38:52: %EASYPY-INFO:     Passx
: 0 2020-03-01T12:38:52: %EASYPY-INFO:     Failed     : 0 2020-03-01T12:38:52:
%EASYPY-INFO:     Aborted    : 0 2020-03-01T12:38:52: %EASYPY-INFO:     Blocked
: 0 2020-03-01T12:38:52: %EASYPY-INFO:     Skipped    : 0 2020-03-01T12:38:52:
```

```
%EASYPY-INFO:     Errored    : 0 2020-03-01T12:38:52: %EASYPY-INFO: 2020-03-
01T12:38:52: %EASYPY-INFO:     TOTAL      : 3 2020-03-01T12:38:52: %EASYPY-INFO:
2020-03-01T12:38:52: %EASYPY-INFO: Success Rate   : 100.00 % 2020-03-01T12:38:52:
%EASYPY-INFO: 2020-03-01T12:38:52: %EASYPY-INFO: +-------------------------------
----------------------------------------+ 2020-03-01T12:38:52: %EASYPY-INFO: |
Task Result Summary                                | 2020-03-01T12:38:52: %EASYPY-
INFO: +------------------------------------------------------------------------
--+ 2020-03-01T12:38:52: %EASYPY-INFO: Task-1: basic_example_script.common_setup
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: Task-1: basic_example_script.Testcase_One
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: Task-1:
basic_example_script.common_cleanup                                PASSED 2020-03-
01T12:38:52: %EASYPY-INFO: 2020-03-01T12:38:52: %EASYPY-INFO: +-------------------
-----------------------------------------------------+ 2020-03-01T12:38:52:
%EASYPY-INFO: |                                Task Result Details
| 2020-03-01T12:38:52: %EASYPY-INFO: +-----------------------------------------
-------------------------------+ 2020-03-01T12:38:52: %EASYPY-INFO: Task-1:
basic_example_script 2020-03-01T12:38:52: %EASYPY-INFO: |-- common_setup
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |   |-- subsection_1
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |   -- subsection_2
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |-- Testcase_One
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |   |-- setup
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |   |-- test_1
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |   |-- test_2
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: |   -- cleanup
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: -- common_cleanup
PASSED 2020-03-01T12:38:52: %EASYPY-INFO:     -- clean_everything
PASSED 2020-03-01T12:38:52: %EASYPY-INFO: Sending report email... 2020-03-
01T12:38:52: %EASYPY-INFO: Missing SMTP server configuration, or failed to
reach/authenticate/send mail. Result notification email failed to send. 2020-03-
01T12:38:52: %EASYPY-INFO: Done!
```

Pro Tip

Use the following command to view your logs locally: pyats logs view. This command automatically opens a page in your web browser displaying the pyATS test results in a GUI format.

```
</code></pre> </details>
```

This is a very simple example that uses the most basic pyATS functionality. There is no actual topology or testbed on which to run network-type tests. However, the output shows you the kind of detailed test log pyATS team creates, including a section-by-section run log of the whole process, from setup to teardown, and appended comprehensive report sections:

```
2020-03-01T12:38:52: %EASYPY-INFO: +-----------------------------------------------
------------------------------+
2020-03-01T12:38:52: %EASYPY-INFO: |                              Task Result
Summary                                |
2020-03-01T12:38:52: %EASYPY-INFO: +-----------------------------------------------
------------------------------+
2020-03-01T12:38:52: %EASYPY-INFO: Task-1: basic_example_script.common_setup
PASSED
```

```
2020-03-01T12:38:52: %EASYPY-INFO: Task-1: basic_example_script.Testcase_One
PASSED
2020-03-01T12:38:52: %EASYPY-INFO: Task-1: basic_example_script.common_cleanup
PASSED
```

Each job run generates an archive `.zip` file, stored by default under your user's home directory, `~/.pyats/archive`. You can list the files and unzip each archive file to view their content (as regular text files), or use the built in web-browser based log viewer, using a localhost web server:

```
pyats logs list
pyats logs view
```

pyATS testbed file

- Define the testbed devices (routers, switches, servers, etc.), subsystems (such as ports, network cards) and their interconnections.
- Establish managerial connections with them, using pyATS' `ConnectionManager` class to create connections and operate upon them. Inside the pyATS topology model, devices are created as objects that include a `connectionmanager` attribute, containing an instance of the `ConnectionManager` class that manages active connections to the real-world device.

The testbed file is an essential input to the rest of pyATS library and ecosystem. It provides information to the framework for loading the right set of library APIs (such as parsers) for each device, and how to effectively communicate to them.

Real testbed files for large topologies can be long, deeply-nested, and complex. A simple `testbed.yaml` file with one device, identified with `<device_ip>` below might look like this example. To run the example, you would need to enter a real IP address for a device that matches the `type` and `os` settings.

Note: This is an example and it will not work with pyATS unless you enter real values for the username, password, and connection IP address.

```
devices:
  router1:
    type: router
    os: nxos
    platform: n9kv
    alias: under_test
    credentials:
      default:
        username: "%ENV{MY_USERNAME}"
        password: "%ENV{MY_PASSWORD}"
    connections:
        cli:
```

```
            protocol: ssh
            ip: "<device_ip>"
```

This example defines a router whose hostname is `router1`, with a supported OS.

- '`platform`' is recommended, and is defined as the machine name (for example, a VM UUID) on which the component is running.
- The file provides default credentials for logging into the device, derived from variables set in your local environment (such as `export MY_USERNAME=username`).
- The file defines the connection method and protocol used to manage the device, and its IP address. pyATS currently communicates via Telnet, SSH, REST, RESTCONF (YANG), and NETCONF (YANG).

To validate that your testbed YAML file meets the pyATS requirements (and conforms to the standard schema), replace the `username`, `password`, and `ip` values, and then run a `pyats validate` command like so:

```
pyats validate testbed testbed.yaml
```

This command checks the content of your file, loads it, and displays any errors in the schema or format.

Note: It is possible to leverage pyATS libraries without a testbed file input, where you can elect to define devices, connections, and other testbed elements programmatically.

pyATS Library: Genie

Genie is the pyATS higher-level library system that provides APIs for interacting with devices, and a powerful CLI for topology and device management and interrogation.

When installed, it adds its features and functionalities into the pyATS framework.

For example, Genie features parsers for a wide range of network operating systems and infrastructure. Parsers are APIs that convert device output into Python structured data. To exercise parsers, enter the pyATS interactive shell. This is effectively the same as Python interpreter/interactive shell, except it provides niche functionalities such as automatically loading your testbed file:

```
pyats shell --testbed-file testbed.yaml
Welcome to pyATS Interactive Shell
==================================
Python 3.6.9 (default, Nov 11 2019, 12:11:42)
[GCC 4.2.1 Compatible Apple LLVM 11.0.0 (clang-1100.0.33.12)]
>>> from pyats.topology import loader
>>> testbed = loader.load('testbed.yaml')
*------------------------------------------------------------------------------
>>>
```

Note: If you are using a pyATS version older than v20.1, the equivalent command is instead `genie shell`.

Now you can access the loaded testbed's devices, establish connectivity, and parse device command outputs like this:

```
# connect to testbed devices in parallel
testbed.connect()
# parse device output from your "router1"
testbed.devices['router1'].parse('show interfaces')
```

The `Device.parse()` API returns the processed structured data (Python dictionary), enabling you to build your own business logic and/or tests on top. To see the list of all available platforms and supported parser commands in Genie library, visit Available Parsers in the Genie documentation.

In addition, it is also possible to exercise the library's functionality through shell CLIs (`pyats` commands). You can interactively extract a comprehensive text description of the configuration and operational states of various protocols, features and hardware information in a given topology. For example:

```
pyats learn conf --testbed-file testbed.yaml --output my_network
```

The underlying Genie mechanism connects in parallel to all the devices defined in the testbed and collects their configurations (`conf`) in a human-readable file (in this case, called `my_network`). The output provides details about network state, including interface setup, VLANs, spanning-tree configuration, and many other features.

Now that the output file exists, it serves as a "gold standard" for this topology's configuration. At any subsequent point in time, if configuration drift seems to have taken hold and broken something, you can run Genie again:

```
pyats learn conf --testbed-file testbed.yaml --output my_broken_network
```

The diff command compares the configurations, quickly exposing differences.

```
pyats diff my_network my_broken_network
```

This API returns a set of diff files, detailing any changes, and letting you quickly discover the cause of problems.

To see the list of "features" pyATS Genie current supports (and can learn), refer to Available Models in the Genie documentation.

Many of pyATS's functions, such as `parse` and `learn`, are exercisable in Python directly (either using interactive shell or in your .py script files, intended for programmers), and through the CLI interface (for non-programmers). You can learn more about this topic the pyATS Getting Started Guide.

Putting it all together

This topic has provided a quick introduction to pyATS and its companion solutions. The next topic introduces VIRL, a Cisco tool for faithfully simulating networks on a server platform, along with some

associated utilities.

Note: Windows platforms are not yet supported for using pyATS.

7.6.3

Lab – Automated Testing Using pyATS and Genie

In this lab, you will explore the fundamentals pyATS and Genie.

You will complete the following objectives:

- Part 1: Launch the DEVASC VM
- Part 2: Create a Python Virtual Environment
- Part 3: Use the pyATS Testing Library
- Part 4: Use Genie to Parse IOS Command Output
- Part 5: Use Genie to Compare Configurations
- Part 6: Lab Cleanup and Further Investigation

Automated Testing Using pyATS and Genie

7.5 Infrastructure as Code

7.7 Network Simulation