Cisco pyATS

Network Test and Automation Solution

Data-driven and reusable testing for modern networks

ciscopress.com

JOHN CAPOBIANCO
DAN WADE

FREE SAMPLE CHAPTER

Cisco pyATS—Network Test and Automation Solution

Data-driven and reusable testing for modern networks

John Capobianco

Dan Wade

Cisco Press

221 River Street

Hoboken, NJ 07030 USA

Cisco pyATS—Network Test and Automation Solution

John Capobianco, Dan Wade

Published by:
Cisco Press

$PrintCode

Library of Congress Control Number: 2024907000

ISBN-13: 978-0-13-803167-1

ISBN-10: 0-13-803167-3

Warning and Disclaimer

This book is designed to provide information about all aspects of Cisco pyATS. Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied.

The information is provided on an "as is" basis. The authors, Cisco Press, and Cisco Systems, Inc. shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the discs or programs that may accompany it.

The opinions expressed in this book belong to the authors and are not necessarily those of Cisco Systems, Inc.

Trademark Acknowledgments

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Feedback Information

At Cisco Press, our goal is to create in-depth technical books of the highest quality and value. Each book is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community.

Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this book, or otherwise alter it to better suit your needs, you can contact us through email at feedback@ciscopress.com. Please make sure to include the book title and ISBN in your message.

We greatly appreciate your assistance.

Please contact us with concerns about any potential bias at https://www.pearson.com/report-bias.html.

GM K12, Early Career and Professional Learning: Soo Kang

Alliances Manager, Cisco Press: Caroline Antonio

Director, ITP Product Management: Brett Bartow

Executive Editor: Nancy Davis

Managing Editor: Sandra Schroeder

Development Editor: Christopher Cleveland

Senior Project Editor: Mandie Frank

Copy Editor: Bart Reed

Technical Editors: Stuart Clark, Charles Greenaway

Editorial Assistant: Cindy Teeters

Designer: Chuti Prasertsith

Composition: codeMantra

Indexer: Timothy Wright

Proofreader: Barbara Mack

Americas Headquarters
Cisco Systems, Inc.
San Jose, CA

Asia Pacific Headquarters
Cisco Systems (USA) Pte. Ltd.
Singapore

Europe Headquarters
Cisco Systems International BV Amsterdam,
The Netherlands

Cisco has more than 200 offices worldwide. Addresses, phone numbers, and fax numbers are listed on the Cisco Website at **www.cisco.com/go/offices.**

About the Authors

John Capobianco has a dynamic and multifaceted career in IT and networking, marked by significant contributions to both the public and private sectors. Beginning his journey in the field as an aluminum factory worker, Capobianco's resilience and dedication propelled him through college, earning a diploma as a Computer Programmer Analyst from St. Lawrence College. This initial phase set the foundation for a career underpinned by continuous learning and achievement, evident from his array of certifications, including multiple Cisco certifications as well as Microsoft certification.

Transitioning from his early educational accomplishments, Capobianco's professional life has spanned over two decades, featuring roles that showcased his technical prowess and strategic vision. His work has significantly impacted both the public and private sectors, including notable positions at the Parliament of Canada, where he served as a Senior IT Planner and Integrator, and at Cisco, where he began as a Developer Advocate. These roles have been instrumental in shaping his perspective on network management and security, leading to his recent advancement into a Technical Leader role in Artificial Intelligence for Cisco Secure, reflecting his commitment to integrating AI technologies for enhancing network security solutions.

In addition to his professional and technical achievements, Capobianco is also an accomplished author. His book *Automate Your Network: Introducing the Modern Approach to Enterprise Network Management*, published in March 2019, encapsulates his philosophy toward leveraging automation for efficient and effective network management. He is dedicated to lifelong learning and professional development, supported by a solid foundation in education and a broad spectrum of certifications, and now aims to share his knowledge with others through this book, YouTube videos, and blogs. John can be found on X using @john_capobianco.

Dan Wade is a Network and Infrastructure Automation Practice Lead at BlueAlly. As part of the Solutions Strategy team at BlueAlly, he is responsible for developing network and infrastructure automation solutions and enabling the sales and consulting teams on delivery of the developed solutions. Solutions may include infrastructure provisioning, configuration management, network source of truth, network observability, and, of course, automated testing and validation. Previous to this role, Dan worked as a consulting engineer with a focus on network automation.

Dan has more than seven years of experience in network automation, having worked with automation tooling and frameworks such as Ansible and Terraform, and Python libraries, including Nornir, Netmiko, NAPALM, Scrapli, and Python SDKs. Dan has been working with pyATS and the pyATS library (Genie) for the past four to five years, which has inspired him to embrace automated network testing. In 2021, Dan contributed to the genieparser library with a new IOS XE parser. He also enjoys creating his own open-source projects focused on network automation. Dan holds two professional-level certifications from Cisco: Cisco DevNet Professional and CCNP Enterprise. He is also a member of the Cisco DevNet 500 and Cisco Champions program.

Dan enjoys sharing knowledge and experience on social media with blog posts and YouTube videos as well as participating in podcast episodes. He's passionate about helping others explore network automation and advocating how network automation can empower, not replace, network engineers. You can find him on social media @devnetdan.

About the Technical Reviewers

Stuart Clark is a senior developer advocate, public speaker, author, and DevNet Expert No. 2022005. Stuart is a sought-after speaker, frequently gracing the stages of industry conferences worldwide, presenting on his expertise in programmability and DevOps methodologies. Passionate about fostering knowledge sharing, he actively creates community content and leads developer communities, empowering others to thrive in the ever-evolving tech landscape. In his previous role as a network engineer, he became obsessed with network automation and became a developer advocate for network automation. He contributed to the Cisco DevNet exams and was part of one of the SME teams that created, designed, and built the Cisco Certified DevNet Expert. He lives in Lincoln, England, with his wife, Natalie, and their son, Maddox. He plays guitar and rocks an impressive two-foot beard while drinking coffee. You can find him on social media @bigevilbeard.

Charles Greenaway, CCIE No. 11226 (R&S, Security, Datacenter), is a field CTO for BT (https://www.bt.com/about/bt). With more than 20 years of data networking experience across LAN/WAN/DC in multiple industry sectors across the globe, he ensures that his customers' use of technology is aligned with their business goals while developing and implementing the technology strategy.

His current focus is helping customers transition toward Global Fabric technologies that provide software-defined underlay and overlay networking to underpin secure multicloud connectivity.

As a member of the DevNet 500 and the Cisco Champions program, Charles promotes the use of programmability and automation to make it accessible to engineers at all levels of skill and experience. He has developed technical content through Greencodemedia Limited and in the public domain at events such as Cisco DevNet Create. Charles is a graduate of Loughborough University, holds a BSc in Computer Science, and lives in the United Kingdom.

Dedications

John: This book is dedicated to my wife and partner of more than 25 years, Michelle. Without her support and encouragement, I would likely still be driving a forklift. J'taime le.

Dan: I would like to dedicate this book to my wonderful wife Hailey and my two amazing kids. They are my foundation and have been patient during the entire writing process. I'd also like to dedicate this book to my parents, who have continued to push me to accomplish whatever I wanted in life. I love you all!

Acknowledgments

John: I would first like to acknowledge what a pivotal role St. Lawrence College has played in my life—first as a student, then as a professor. Thank you to Donna Graves, Janis Michael, and, of course, rest in peace, Carl Davis. To everyone I've ever worked with in my career from Empire Life to the Parliament of Canada, thank you. I am very proud of what we accomplished together and for the confidence you had in me to build, support, evolve, and, ultimately, automate your networks. Thanks also to Cisco for embracing me completely as one of your own—I've never had such a supportive culture.

Thanks as well to everyone involved with the publishing of this book, from Nancy Davis and Chris Cleveland and the Pearson team, to our editors Stuart Clark and Charles Greenaway for their dedication to the project, and last but not least, to Dan Wade for co-authoring the book. From joint live streams to collaborating on this book, I am really proud to call you a friend.

Finally, to JB and Siming, for inviting me to a private pyATS crash course. You have both been so giving and have provided me real guidance and direction and turned me onto Python. Thank you both.

Dan: First, I'd like to thank the Art of Network Engineering (AONE) community, specifically AJ Murray, for encouraging me to begin blogging and creating my own brand. I can confidently say there would be no DevNet Dan without AONE! I'd also like to thank NetCraftsmen for taking a chance on me in the beginning of my consulting career. It was my first time working in the consulting space, and they've consistently guided me to success. Thank you Terry, Shaffeel, Robert, Bill, Joel, and John for continuing to encourage me and push me to grow professionally.

I would like to thank Nancy Davis for giving me the opportunity to pursue this project. She continues to encourage and support me to pursue creative opportunities. A big thank you to Chris Cleveland, development editor, for providing the best support developing the book, and to Stuart and Charles for their unbiased and honest technical review of the book and its contents. I'd also like to thank my wonderful co-author John. It has been a pleasure working and writing this phenomenal book with him!

Finally, thanks to all the content creators, trainers, and authors who have influenced my narration style and contributed to my constant learning of networking and software development.

Contents at a Glance

Reader Services

Register your copy at www.ciscopress.com/title/ISBN for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to www.ciscopress.com/register and log in or create an account*. Enter the product ISBN 9780138031671 and click Submit. When the process is complete, you will find any available bonus content under Registered Products.

*Be sure to check the box that you would like to hear from us to receive exclusive discounts on future editions of this product.

Contents

Command Syntax Conventions

The conventions used to present command syntax in this book are the same conventions used in the IOS Command Reference. The Command Reference describes these conventions as follows:

- **Boldface** indicates commands and keywords that are entered literally as shown. In actual configuration examples and output (not general command syntax), boldface indicates commands that are manually input by the user (such as a **show** command).
- *Italic* indicates arguments for which you supply actual values.
- Vertical bars (|) separate alternative, mutually exclusive elements.
- Square brackets ([ ]) indicate an optional element.
- Braces ({ }) indicate a required choice.
- Braces within brackets ([{ }]) indicate a required choice within an optional element.

Foreword

In late 2013, I found myself seated at the end of a restaurant table in San Jose, celebrating the success of our latest Tcl language–based test automation feature release. Tibor Fabry-Asztalos, our visionary senior director, raised his glass in a toast: "We need to look at our next goal. It's time to transition to Python-based automation." With that, he gazed to his left, where coincidentally his chain of reports was seated in order—and each person looked further to their left, until there was just me, the final link in the chain, entrusted to shoulder that responsibility.

And so pyATS was born.

After two decades of Tcl/Expect-based automation and testing in Cisco, the call for a more modern, natively object-orienting infrastructure was undeniable—one that could scale forward, lower the barrier for adoption, and attract new talents as Tcl expertise waned.

2024 marks the 10-year anniversary for pyATS. Originally introduced as an internal testing solution, its 2017 public launch through Cisco DevNet marked a definitive, transformative moment. It enabled closer collaboration between Cisco engineering, customers, and their network engineers, unlocking a plethora of opportunities and use cases. Around that time, NetDevOps was in its infancy, and network engineers were seeking for their next career breakthrough. pyATS was ready just around the corner.

Rarely does one find themselves at the helm of opportunity to shape the next decade of network automation, a chance to redefine the landscape of network testing, and influence the careers of countless network engineers. It has been an exciting journey, filled with dedication, perseverance, and innovation. Most importantly, we took pride in what we have created and accomplished.

Looking back, could we have done better? Absolutely. Along the way, mistakes were made, and compromises became necessary. But as someone special to me once said, "every decision you make in life [sic] is always the best that you could, based on the limited knowledge you had at that time." We, the pyATS development team, gave it our best, and the community echoed positively.

It has been a privilege and an honor to be able to stand at the precipice of a new chapter in the history of test automation at Cisco. A heartfelt thank-you goes to our team, our community members, and everyone who supported us along the way. As pyATS continues to evolve, my sincerest wishes for its continued momentum and enduring legacy.

```
>>> from pyats import awesome
```

—Siming Yuan, pyATS Founder, Architect and Lead Developer

Reflecting on the inception of pyATS, it's astounding to see the journey from an ambitious project within Cisco engineering to a cornerstone of network automation. Born from the challenges we faced daily, it quickly grew beyond the initial scope, demonstrating the power of innovative solutions in a rapidly evolving field. I am filled with gratitude for the brilliant minds I worked with and the community that has grown around these tools. Your enthusiasm and support have been the driving force behind its success.

Now, looking back, I see the legacy of pyATS not just in the technical achievements, but in the community and collaboration it fostered. It has been a privilege to contribute to this chapter of network engineering, and I am proud of what we accomplished together.

A special thanks to all the pyATS team members who have worked on it. It wouldn't have been possible without you. Thank you to everyone who has joined us on this remarkable journey. Your contributions have made all the difference.

—Jean-Benoit Aubin, Lead Developer and Architect, pyATS

Introduction

This book was written to explore the powerful capabilities of automated network testing with the Cisco pyATS framework. Network testing and validation is a low-risk, yet powerful domain in the network automation space. This book is organized to address the multiple features of pyATS and the pyATS library (Genie). Readers will learn why network testing and validation are important, how pyATS can be leveraged to run tests against network devices, and how to integrate pyATS into larger workflows using CI/CD pipelines and artificial intelligence (AI).

Goals and Objectives

This book touches on many aspects of network automation, including device configuration, data parsing, APIs, parallel programming, artificial intelligence, and, of course, automated network testing. The intended audience for this book is network professionals and software developers wanting to learn more about the pyATS framework and the benefits of automated network testing. The audience should be comfortable with Python, as pyATS is built with the Python programming language.

Candidates who are looking to learn pyATS as it relates to the Cisco DevNet Expert Lab exam will find the use cases and examples throughout the book valuable for exam preparation.

How This Book Is Organized

Chapter 1, **"Foundations of NetDevOps"**: This chapter introduces NetDevOps, outlining its benefits and how it merges with software development methodologies to enhance network automation. We compare key automation tools and detail the modern network engineer's toolkit, setting the stage for applying NetDevOps in practice.

Chapter 2, **"Installing and Upgrading pyATS"**: The chapter shows how to install and upgrade pyATS and the pyATS library using Python package management tools and built-in pyATS commands.

Chapter 3, "Testbeds": This chapter covers YAML's basics, explores the concept of a testbed, and examines device connection abstractions. We discuss methods for testbed validation, the creation of dynamic testbeds, and how intent-based networking integrates with extended testbeds, providing a roadmap for their practical application.

Chapter 4, **"AEtest Test Infrastructure"**: This chapter is one of the key chapters in this book. It goes in depth and reviews the different components that make up AEtest, the testing infrastructure that is the core of pyATS. Everything from defining testcases and individual test sections to running testscripts is covered in this chapter. After reading this chapter, you'll understand how to introduce test inputs and parameters, define test sections, control the flow of test execution, and review test results with the built-in reporting features.

Chapter 5, "pyATS Parsers": This chapter delves into pyATS parsers, emphasizing vendor-neutral automation strategies. It covers the essentials of pyATS learn and parse features, techniques for CLI parsing, and parsing with Python. Additionally, we explore how to perform dictionary queries and analyze differentials, equipping you with the necessary skills for effective network data handling.

Chapter 6, "Test-Driven Development": This chapter introduces test-driven development (TDD), its application in network automation, and an overview of pyATS. It further explores the pyATS framework, setting the foundation for incorporating TDD practices into network management.

Chapter 7, "Automated Network Documentation": This chapter explores automated network documentation, beginning with an introduction to pyATS jobs. It details executing pyATS jobs from the command-line interface (CLI), interpreting CLI logs, and utilizing the pyATS logs HTML viewer for enhanced analysis. We also delve into Jinja2 templating for document creation, culminating in the generation of business-ready documents.

Chapter 8, "Automated Network Testing": This pivotal chapter delves into automated network testing, the core focus of the book. It outlines a strategic approach to network testing, including software version testing, interface testing, neighbor testing, and reachability testing. Additionally, we explore intent-validation testing and feature testing, essential components for ensuring network reliability and performance.

Chapter 9, "pyATS Triggers and Verifications": This chapter reviews how to use triggers and verifications using the Genie Harness. Triggers and verifications allow you to build dynamic testcases, with a low-code approach, that can change with your network requirements.

Chapter 10, "Automated Configuration Management": In this chapter we will look at how to generate intent-based configuration using data models, Jinja2 templates, and Genie Conf objects. In addition to generating configurations, we will see how to push configuration to network devices using a file transfer server, Genie Conf objects, and pyATS device APIs.

Chapter 11, "Network Snapshots": This chapter looks at how to profile the network by creating and comparing snapshots of the network. Network snapshots can be helpful when you're troubleshooting a network issue or just learning about the network's operating state at a point in time.

Chapter 12, "Recordings, Playbacks, and Mock Devices": This chapter introduces pyATS recordings, covering the recording of pyATS jobs and the playback of these recordings. It explains how to create mock devices and simulate device interactions through the mock device CLI, offering practical insights into testing without the need for live network equipment.

Chapter 13, "Working with Application Programming Interfaces (API)": This chapter focuses on working with pyATS APIs, detailing the pyATS API framework, REST connector, YANG connector, and gNMI. It provides insights into how these tools and protocols can be utilized for efficient network automation and management through API interactions.

Chapter 14, "Parallel Call (pcall)": Testing in pyATS can be sped up using parallel processing (parallelism). In this chapter, we review the differences between parallelism and concurrency using asynchronous programming. Parallel call (pcall) in pyATS enables parallel execution and is built on the multiprocessing package in the Python standard library.

Chapter 15, "pyATS Clean": In this chapter, you will see how pyATS can reset devices during or after testing using the pyATS Clean feature.

Chapter 16, "pyATS Blitz": In this chapter, we will review pyATS Blitz, which creates a low-code approach to building pyATS testcases using YAML syntax.

Chapter 17, "Chatbots with Webex": This chapter explores integrating pyATS with Webex, including pyATS job and health check integrations. It delves into using Adaptive Cards within Webex for interactive content and outlines methods for setting up customized job notifications, enhancing communication and monitoring in network operations.

Chapter 18, "Running pyATS as a Container": This chapter introduces the concept of containers, focusing on the pyATS official Docker container. It guides you through the pyATS image builder and details the process of building a pyATS image from scratch, offering a comprehensive approach to deploying pyATS as a containerized application.

Chapter 19, "pyATS Health Check": This chapter dives into the different health checks that run to ensure devices under testing are operating correctly. Built-in health checks include checking CPU, memory, logging, and the presence of core dump files to ensure devices haven't malfunctioned or crashed during testing.

Chapter 20, "XPRESSO": This section covers pyATS XPRESSO, starting with installation instructions. It provides a beginner's guide to getting started with XPRESSO and details on running pyATS jobs within the XPRESSO environment, facilitating an easy entry into utilizing this powerful tool.

Chapter 21, "CI/CD with pyATS": The concept of CI/CD is a common practice in software development to build and test code before it's pushed to production. In this chapter, we see how to use multiple network automation tools, including GitLab, Ansible, and pyATS, to apply CI/CD practices when pushing configuration changes to the network.

Chapter 22, "Robot Framework": In this chapter, we review Robot Framework, an open-source test automation framework. Robot Framework allows you to use English-like keywords to define testcases. After we review Robot Framework, we see how the pyATS libraries—Unicon, pyATS, and the pyATS library (Genie)—are integrated into Robot Framework by providing test libraries that include keywords to interact with network devices and define testcases.

Chapter 23, "Leveraging Artificial Intelligence in pyATS": This chapter explores the integration of pyATS with artificial intelligence, focusing on leveraging the OpenAI API for enhanced network automation. It discusses the use of retrieval augmented generation (RAG) with Langchain for intelligent data handling and introduces rapid prototyping with Streamlit, showcasing the potential for AI to revolutionize network management processes.

Appendix A, "Writing Your Own Parser": This appendix covers how to contribute to the genieparser library (https://github.com/CiscoTestAutomation/genieparser) by creating a new parser for a Cisco IOS XE **show** command.

Appendix B, "Secret Strings": This appendix covers how to protect the sensitive data in your testbed.yaml files through secret strings.

Credits

Figure 1.1: chinnappa/123RF

Figures 5.2a, 5.7-5.9, 5.13, 5.14, 13.2a, 21.1, AppA 1, 2, 4-7: GitHub, Inc

Figures 5.15-5.17, 7.24-7.34: Microsoft Corporation

Figure 20.26a: Jenkins

Figures 21.3, 21.4: GitLab B.V.

Figures 23.3-23.6: Snowflake Inc

This page intentionally left blank

Chapter 9

pyATS Triggers and Verifications

Automated network tests face challenges in achieving futureproofing due to the dynamic nature of networks and their evolving requirements. The pyATS library (Genie) provides the Genie Harness to execute network tests with dynamic and robust capabilities. The Genie Harness is part of the pyATS library (Genie), which is built on the foundation of pyATS. It introduces the ability for engineers to create dynamic, event-driven tests with the use of processors, triggers, and verifications. In this chapter, the following topics are going to be covered:

- Genie objects
- Genie Harness
- Triggers
- Verifications

The Genie Harness can be daunting for new users of the pyATS library (Genie), as there are many configuration options and parameters. The focus of this chapter will be to provide an overview of Genie Harness and its features and wrap up with a focus on triggers and verifications. By the end of the chapter, you'll understand the powerful capabilities of Genie Harness and how to write and execute triggers and verifications.

Genie Objects

One of the hardest parts of network automation is figuring out how to normalize and structure the data returned from multiple network device types. For example, running commands to gather operational data (CPU, memory, interface statistics, routing state, and so on) from a Cisco IOS XE switch, IOS XR router, and an ASA firewall may have different, but similar, output. How do we account for the miniscule differences across

the different outputs? The pyATS library (Genie) abstracts the details of the parsed data returned from devices by creating network OS-agnostic data models. Two types of Genie objects represent these models: Ops and Conf. In the following sections, you'll dive into the details of each object.

Genie Ops

As you may be able to guess from the name, the Genie Ops object learns everything about the operational state of a device. The operational state data is represented as a structured data model, or feature. A *feature* is a network OS-agnostic data model that is created when pyATS "learns" about a device. Multiple commands are executed on a device and the output is parsed and normalized to create the structure of the feature. You can access the learned feature data by accessing **<feature>.info**. To learn more information about the available models, check out the Genie models documentation (https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/models). Example 9-1 shows how to instantiate an Ops object for BGP and learn the BGP feature on a Cat8000v device. Figure 9-1 shows the associated output.

Example 9-1 *Genie Ops Object*

```
from genie import testbed
from genie.libs.ops.bgp.iosxe.bgp import Bgp
import pprint

# Load Genie testbed
testbed = testbed.load(testbed="testbed.yaml")

# Find the device using hostname or alias
uut = testbed.devices["cat8k-rt2"]
uut.connect()

# Instantiate the Ops object
bgp_obj = Bgp(device=uut)

# This will send many show commands to learn the operational state
# of BGP on this device
bgp_obj.learn()

pprint.pprint(bgp_obj.info)
```

```
{'instance': {'default': {'bgp_id': 65001,
                          'vrf': {'default': {'cluster_id': '10.254.1.2',
                                              'neighbor': {'172.16.1.1': {'address_family': {'ipv4 unicast': {'bgp_table_version': 2,
                                                                                                              'path': {'memory_usage': 136,
                                                                                                                       'total_entries': 1},
                                                                                                              'prefixes': {'memory_usage': 248,
                                                                                                                           'total_entries': 1},
                                                                                                              'routing_table_version': 2,
                                                                                                              'total_memory': 680}},
                                                                          'bgp_neighbor_counters': {'messages': {'received': {'keepalives': 0,
                                                                                                                              'notifications': 0,
                                                                                                                              'opens': 0,
                                                                                                                              'updates': 0},
                                                                                                                 'sent': {'keepalives': 0,
                                                                                                                          'notifications': 0,
                                                                                                                          'opens': 0,
                                                                                                                          'updates': 0}}},
                                                                          'bgp_session_transport': {'connection': {'last_reset': 'never',
                                                                                                                   'state': 'Active'}},
                                                                          'bgp_version': 4,
                                                                          'remote_as': 65000,
                                                                          'session_state': 'Active',
                                                                          'shutdown': False}}}}}}}}
```

Figure 9-1 *Genie Ops Object Output*

Genie Conf

The Genie Conf object allows you to take advantage of Python's object-oriented programming (OOP) approach by building logical representations of devices and generating configurations based on the logical device and its attributes. Just like Genie Ops, the structure of Genie Conf objects is based on the feature models. The best way to describe the Conf object is by example. Example 9-2 shows how to build a simple network interface object for an IOS XE device. You'll notice that the code is very comprehensive, specifically for a network engineer, and will even generate the configuration!

Example 9-2 *Genie Conf Object*

```
from genie import testbed
from genie.libs.conf.interface.iosxe.interface import Interface
import pprint



# Load Genie testbed
testbed = testbed.load(testbed="testbed.yaml")

# Find the device using hostname or alias
uut = testbed.devices["cat8k-rt2"]

# Instantiate the Conf object
interface_obj = Interface(device=uut, name="Loopback100")

# Add attributes to the Interface object
interface_obj.description = "Managed by pyATS"
interface_obj.ipv4 = "1.1.1.1"
interface_obj.ipv4.netmask = "255.255.255.255"
interface_obj.shutdown = False
```

```
# Build the configuration for the interface
print(interface_obj.build_config(apply=False))



! Code Execution Output
interface Loopback100
 description Managed by pyATS
 ip address 1.1.1.1 255.255.255.255
 no shutdown
 exit
```

You may notice the last line of code actually builds the necessary configuration for you! To go one step further, you can add the interface object to a testbed device and push the configuration to a device. The Conf object allows you to drive the configuration of network devices with Python, which takes you one step further into the world of automation.

Genie Harness

The Genie Harness is structured much like a pyATS testscript. There are three main sections: Common Setup, Triggers and Verifications, and Common Cleanup. The Common Setup section will connect to your devices, take a snapshot of current system state, and optionally configure the devices, if necessary. The Triggers and Verifications section, which you will see later in the chapter, will execute the triggers and verifications to perform tests on the devices. This is where the action happens! The Common Cleanup section confirms that the state of the devices is the same as their state in the Common Setup section by taking another snapshot of the current system state and comparing it with the one captured in the Common Setup section.

gRun

The first step to running jobs with Genie Harness is creating a job file. Job files are set up much like a pyATS job file. Within the job file there's a main function that is used as an entry point to run the job(s). However, instead of using **run()** within the main function to run the job, you must use **gRun()**, or the "Genie Run" function. This function is used to execute pyATS testscripts with additional arguments that provide robust and dynamic testing. Datafiles are passed in as arguments, which allows you to run specific triggers and verifications. Example 9-3 shows a Genie job file from the documentation that runs one trigger and one verification.

Example 9-3 *Genie Harness – Job file*

```
from genie.harness.main import gRun

def main():
        # Using built-in triggers and verifications
        gRun(trigger_uids=["TriggerSleep"],
        verification_uids=["Verify_IpInterfaceBrief"])
```

The trigger and verification names in Example 9-3 are self-explanatory: sleep for a specified amount of time and parse and verify the output of **show ip interface brief** command. Each trigger and verification is part of the pyATS library. If these were custom-built triggers or if a testbed device did not have the name or alias of "uut" in your testbed file, you would need to create a mapping datafile. A mapping datafile is used to create a relationship, or mapping, between the devices in a testbed file and Genie. It's required if you want to control multiple connections and connection types (CLI, XML, YANG) to testbed devices. By default, Genie Harness will only connect to the device with the name or alias of "uut," which represents "unit under testing." The uut name/alias requirement allows Genie Harness to properly load the correct default trigger and verification datafiles. Otherwise, you must include a list of testbed devices to the triggers/verifications in the respective datafile. If this doesn't make sense, don't worry; we will touch on datafiles and provide examples further in the chapter that should help provide clarity.

To wrap up the Genie Harness example, the Genie job is run with the same Easypy runtime environment used to run pyATS jobs. To run the Genie job from the command line, enter the following:

```
(.venv)dan@linux-pc# pyats run job {job file}.py --testbed-file
{/path/to/testbed}
```

Remember to have your Python virtual environment activated! In the next section, we will jump into datafiles and how they are defined.

Datafiles

The purpose of a datafile is to provide additional information about the Genie features (triggers, verifications, and so on) you would like Genie Harness to run during a job. For example, the trigger datafile may specify which testbed devices a trigger should run on in the job. There are many datafiles available in Genie Harness. However, many of them are optional and are only needed if you're planning to modify the default datafiles provided. The default datafiles can be found at the following path:

```
$VIRTUAL_ENV/lib/python<version>/site-packages/genie/libs/sdk/genie_
yamls
```

Within the genie_yamls directory, you'll find default datafiles that apply to all operating systems (OSs) and others that are OS-specific. These default datafiles are only implicitly

activated when a testbed device has either a name or alias of uut. If there isn't a testbed device with that name or alias, the default datafile will not be implicitly passed to that job. I would highly recommend checking out (not editing) the default datafiles. If you'd like to edit one, you may create a new datafile in your local directory and extend the default one—but don't jump too far ahead yet! This topic will be covered later in the chapter. Here's a list of the different datafiles that can be passed to **gRun**:

- Testing datafile
- Mapping datafile
- Verification datafile
- Trigger datafile
- Subsection datafile
- Configuration datafile
- PTS datafile

Each datafile serves a purpose to a specific Genie Harness feature, but one only needs to be specified if you are deviating from the provided default datafile. For example, the Profile the System (PTS) default datafile only specifies to run on the testbed device with the alias uut. If you would like it to run on more devices, you'll need to create a **pts_datafile.yaml** file that maps devices to the device features you want profiled by PTS and include the **pts_datafile** argument to **gRun**.

Device Configuration

Applying device configuration is always a hot topic when it comes to network automation. In the context of pyATS and the pyATS library (Genie), the focus is on applying (and reverting) the configuration during testing. In many cases, you'll want to test a feature by configuring it on a device, testing its functionality, and removing the configuration before the end of testing. The pyATS library (Genie) provides many ways to apply configuration to devices. Here are some of the options you have to configure a network device during testing:

- Manual configuration before testing begins (not recommended).
- Automatically apply the configuration to the device in the Common Setup and Common Cleanup sections with TFTP/SCP/FTP. A config.yaml file can be provided to the **config_datafile** argument of **gRun**, which specifies the configuration to apply.
- Automatically apply the configuration to the device in the Common Setup and Common Cleanup sections using Jinja2 template rendering. The Jinja2 template filename will be passed to **gRun** using the **jinja2_config** argument and the device variables will be passed as key-value pairs using the **jinja2_arguments** argument.

You will dive into Jinja2 templates further and how to use them to generate configurations in Chapter 10, "Automated Configuration Management." For now, just understand that you can standardize the configuration being pushed to the network devices under testing using configuration templates and a template rendering engine (Jinja2) to render the templates with device variables, resulting in complete configuration files.

All configurations should be built using the **show running** style, which means you create your configuration files how they would appear when you view a device's configuration using the **show running-config** command. This differs from how you would configure a device interactively via SSH using the **configure terminal** approach.

After the devices under testing have been configured, the pyATS library learns the configuration of the devices via the check_config subsection. The check_config subsection runs twice: once during Common Setup and another time during Common Cleanup. It collects the **running-config** of each device in the topology and compares the two configuration "snapshots" to ensure the configuration remains the same before and after testing.

PTS (Profile the System)

Earlier in this chapter, you saw examples of device features that can be learned (for example, BGP) during testing. This is made possible by the network OS-agnostic models built into pyATS. These models create the foundation for building reliable data structures and provide the ability to parse data from the network.

PTS provides the ability to "profile" the network during testing. PTS creates profile snapshots of each feature in the Common Setup and Common Cleanup sections. PTS can learn about all the device features, or a specific list of device features can be provided as a list to the **pts_features** argument of **gRun**. Example 9-4 shows how **gRun** is called in a job file with a list of features passed to the **pts_features** argument.

Example 9-4 *PTS Feature*

```
from genie.harness.main import gRun

def main():
        # Profiling built-in features (models) w/o the PTS datafile
        gRun(pts_features=["bgp", "interface"])
```

Along with having the ability to profile a subset of features/device commands, PTS, by default, will only run on the device with the device alias uut. To have more devices profiled by PTS, you'll need to supply a pts_datafile.yaml file. The datafile can provide a list of devices to profile and describe specific attributes to ignore in the output when comparing snapshots (such as timers, uptime, and dates). Example 9-5 shows a PTS datafile, and Example 9-6 shows the updated **gRun** call, with the **pts_datafile** argument included.

Example 9-5 *PTS Datafile – ex0906_pts_datafile.yaml*

```
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(pts)}"

bgp:
  devices: ["cat8k-rt1", "cat8k-rt2"]
  exclude:
    - up_time

interface:
  devices: ["cat8k-rt1", "cat8k-rt2"]
```

Example 9-6 *PTS Datafile Argument*

```
from genie.harness.main import gRun

def main():
         # Profiling built-in features (models) w/ the PTS datafile
         gRun(pts_features=["bgp", "interface"],
         pts_datafile="ex0906_pts_datafile.yaml")
```

PTS Golden Config

PTS profiles the operational state of the network devices under testing, but how do we know the state is what we expect? PTS provides a "golden config" snapshot feature that compares the profiles learned by PTS to what is considered the golden snapshot. Each job run generates a file named pts that is saved to the pyATS archive directory of the job. Any PTS file can be moved to a fixed location and used as the golden snapshot. Like the **pts_datafile** argument, the **pts_golden_config** argument can be passed to **gRun**, which points to the golden PTS snapshot used to compare against the current test run snapshots.

There's a lot to digest with Genie Harness as well as a lot of different options, and an understanding of these features and when to use them is critical. In the following sections, we will turn our attention to triggers and verifications. Let's take a look at verifications first, as triggers rely on them to perform properly.

Verifications

A verification runs one or multiple commands to retrieve the current state of the device. The main purpose of a verification is to capture and compare the operational state before and after a trigger, or set of triggers, performs an action on a device. The state of the device can be retrieved via the multiple connection types offered by the pyATS library (CLI, YANG, and so on). Verifications typically run in conjunction with triggers to verify the trigger action did what it was supposed to do and to check for unexpected results, such as changes to a feature you didn't initiate.

Verification Types

Verifications can be broken down into two types: global and local. The difference between the two is related to scoping and when each verification type runs within a script.

- **Global verifications:** Global verifications are used to capture a snapshot of a device before a trigger is executed. Global verifications run immediately after the Common Setup section and before a trigger in a script. If more than one trigger is executed, subsequent snapshots are captured before and after each trigger using the same set of verifications.
- **Local verifications:** Local verifications are independent of global verifications and run as subsections within a trigger. More specifically, a set of snapshots is taken before a trigger action and a subsequent set is taken after to compare to the first one. Local verifications confirm the trigger action did what it was supposed to do (configure/unconfigure, shut/no shut, and so on).

Figure 9-2 shows where and when the different verification types run in a Genie job.

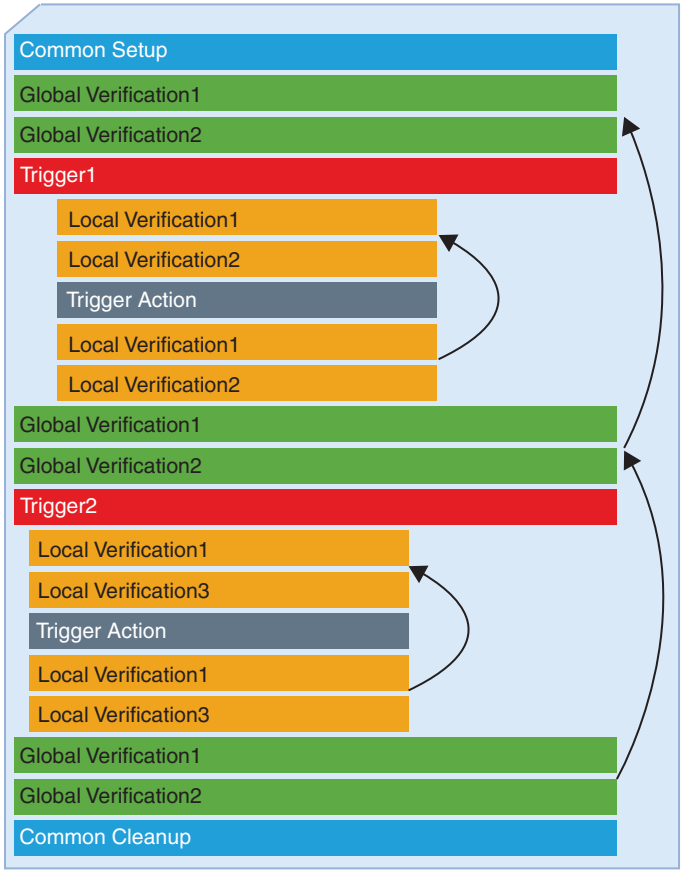

Figure 9-2 *Verification Execution*

Verification Datafile

A verification datafile is used to customize the execution of built-in or custom verifications. Like other datafiles, verification_datafile.yaml must be provided to **gRun** using the **verification_datafile** argument. Example 9-7 shows a verification datafile that extends the default verification datafile (via the **extends:** key) and overrides the default setting of connecting to only the "uut" device (via the **devices:** key) for the **Verify_Interfaces** verification. If you wanted to change the list of devices to connect to for another verification, you would need to add that verification in the datafile.

Example 9-7 *Verification Datafile – ex0908_verification_datafile.yaml*

```
# Extend default verification datafile to inherit the required keys
# (class, source, etc.) per the verification datafile schema
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(verification)}"

Verify_Interfaces:
  devices: ["cat8k-rt1", "cat8k-rt2"]
```

In order to run the verifications listed in the datafile, or any other built-in verifications, you'll need to include them in the **verification_uids** argument to **gRun**. Example 9-8 shows how to run the Verify_Interfaces verification with the verification datafile from Example 9-7.

Example 9-8 *gRun – Verifications*

```
from genie.harness.main import gRun



def main():
         gRun(
         verification_uids=["Verify_Interfaces"],
         verification_datafile="ex0908_verification_datafile.yaml"
         )
```

Writing a Verification

The process in which a feature is verified during testing may be different for different use cases. If a built-in verification does not suffice, the pyATS library (Genie) allows you to create your own verification.

There are several ways to create your own verification. You can use a Genie Ops feature (model), a parser, or callable. For the Genie Ops feature and parser options, you can use an existing model or parser, or you can create your own. The last option, using a callable, is discouraged, as it isn't OS-agnostic and does not provide extensibility to use different

management interfaces (CLI, YANG, and so on). Example 9-9 shows a custom verification built with the **show bgp all** parser. Take note of the list of excluded values from the parsed data (found under the **exclude:** key). The reason is because many of these values are dynamic (such as timers, counters, and so on) and are almost guaranteed to be different between snapshots. Remember, if a parsed value is different between snapshots, the verification will fail.

Example 9-9 *Custom Verification – ex0910_verification_datafile.yaml*

```
# Local verification datafile that already extends the default datafile
extends: verification_datafile.yaml

Verify_Bgp:
    cmd:
        class: show_bgp.ShowBgpAll
        pkg: genie.libs.parser
    context: cli
    source:
        class: genie.harness.base.Template
    devices: ["cat8k-rt1", "cat8k-rt2"]
    iteration:
        attempt: 5
        interval: 10
    exclude:
    - if_handle
    - keepalives
    - last_reset
    - reset_reason
    - foreign_port
    - local_port
    - msg_rcvd
    - msg_sent
    - up_down
    - bgp_table_version
    - routing_table_version
    - tbl_ver
    - table_version
    - memory_usage
    - updates
    - mss
    - total
    - total_bytes
    - up_time
```

```
    - bgp_negotiated_keepalive_timers
    - hold_time
    - keepalive_interval
    - sent
    - received
    - status_codes
    - holdtime
    - router_id
    - connections_dropped
    - connections_established
    - advertised
    - prefixes
    - routes
    - state_pfxrcd
```

To run the custom verification, you follow the same process as running any other verification. Pass the verification datafile with the custom verification to **gRun** via the **verification_datafile** argument and add the custom verification name to the list of verifications in the **verification_uids** argument. Example 9-10 shows the updated **gRun** call with the custom verification name and datafile. Remember, the custom verification datafile (Example 9-9) extends the original verification datafile (Example 9-7), which essentially inherits all the built-in verifications from the pyATS library (Genie).

Example 9-10 *gRun – Custom Verification*

```
from genie.harness.main import gRun



def main():
         gRun(
         verification_uids=["Verify_Interfaces", "Verify_Bgp"],
         verification_datafile="ex0910_verification_datafile.yaml"
         )
```

Triggers

Triggers perform a specific action, or a sequence of actions, on a device to alter its state and/or configuration. As examples, actions may include adding/removing parts of a configuration, flapping protocols/interfaces, or performing high availability (HA) events such as rebooting a device. The important part to understand is that triggers are what alter the device during testing.

The pyATS library (Genie) has many prebuilt triggers available for Cisco IOS/IOS XE, NX-OS, and IOS XR. All prebuilt triggers are documented, describing what happens

when the trigger is initiated and what keys/values to include in the trigger datafile specifically for that trigger. For example, the **TriggerShutNoShutBgpNeighbors** trigger performs the following workflow:

1. Learn BGP Ops object and verify it has "established" neighbors. If there aren't any "established" neighbors, skip the trigger.
2. Shut the BGP neighbor that was learned from step 1 with the BGP Conf object.
3. Verify the state of the learned neighbor(s) in step 2 is "down."
4. Unshut the BGP neighbor(s).
5. Learn BGP Ops again and verify it is the same as the BGP Ops snapshot in step 1.

As you might recall from earlier in this chapter, the Genie Ops object represents a device/feature's operational state via a Python object, and the Genie Conf object represents a feature, as a Python object, that can be configured on a device. The focus of the Conf object is *what* feature you want to apply on the device, not *how* to apply it per device (OS) platform. This allows a network engineer to focus on the network features being tested and not on the low-level details of how the configuration is applied.

Now that there's a general understanding of what triggers do, let's check out how they can be configured using a trigger datafile.

Trigger Datafile

As with other features of the pyATS library (Genie), to run triggers, there needs to be a datafile—more specifically, a trigger datafile (trigger_datafile.yaml). The pyATS library provides a default trigger datafile found in the same location as all the other default datafiles, discussed earlier in the chapter. However, if you want to customize any specific trigger settings, such as what devices or group of devices to run on during testing (any device besides uut), or to run a custom trigger, you'll need to create your own trigger datafile. A complete example can be found at the end of the chapter that includes both triggers and verifications, but let's focus now on just triggers in a brief example. Example 9-11 shows a custom trigger file that flaps the OSPF process on the targeted devices (iosv-0 and iosv01). Example 9-12 shows how to include the appropriate trigger and trigger datafile in the list of arguments to **gRun**.

Example 9-11 *Trigger Datafile – ex0912_trigger_datafile.yaml*

```
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(trigger)}"

# Custom trigger - created in Example 9-14
TriggerShutNoShutOspf:
# source imports the custom trigger, just as you would any other Python class
  source:
    class: ex0915_custom_trigger.ShutNoShutOspf
  devices: ["iosv-0", "iosv-1"]
```

Example 9-12 *gRun – Triggers and Trigger Datafile*

```
from genie.harness.main import gRun



def main():
         gRun(
         trigger_uids=["TriggerShutNoShutOspf"],
         trigger_datafile="ex0912_trigger_datafile.yaml"
         )
```

Trigger Cluster

The last neat trigger feature to cover is the ability to execute a group of multiple triggers and verifications in one cluster trigger. First, a trigger datafile must be created with the list of triggers and verifications, the order in which to run them, and a list of testbed devices to run them against. Example 9-13 shows a trigger datafile configured for a trigger cluster and the accompanying test results if it was run.

Example 9-13 *Trigger Cluster*

```
TriggerCombined:
    sub_verifications: ['Verify_BgpVrfAllAll']
    sub_triggers: [ 'TriggerSleep', 'TriggerShutNoShutBgp']
    sub_order: ['TriggerSleep', 'Verify_BgpVrfAllAll',
    'TriggerSleep','TriggerShutNoShutBgp','Verify_BgpVrfAllAll']
    devices: ['uut']

-- TriggerCombined.uut                                          PASSED
   |-- TriggerSleep_sleep.1                                     PASSED
   |-- TestcaseVerificationOps_verify.2                         PASSED
   |-- TriggerSleep_sleep.3                                     PASSED
   |-- TriggerShutNoShutBgp_verify_prerequisite.4               PASSED
   |   |-- Step 1: Learning 'Bgp' Ops                           PASSED
   |   |-- Step 2: Verifying requirements                       PASSED
   |   '-- Step 3: Merge requirements                           PASSED
   |-- TriggerShutNoShutBgp_shut.5                              PASSED
   |   '-- Step 1: Configuring 'Bgp'                            PASSED
   |-- TriggerShutNoShutBgp_verify_shut.6                       PASSED
   |   '-- Step 1: Verifying 'Bgp' state with ops.bgp.bgp.Bgp   PASSED
   |-- TriggerShutNoShutBgp_unshut.7                            PASSED
   |   '-- Step 1: Unconfiguring 'Bgp'                          PASSED
   |-- TriggerShutNoShutBgp_verify_initial_state.8              PASSED
   |   '-- Step 1: Verifying ops 'Bgp' is back to original state PASSED
   '-- TestcaseVerificationOps_verify.9                         PASSED
```

You may notice that the triggers have accompanying local verifications that run before and after the trigger is run to ensure the action was actually taken against the device. This is the true power of triggers. One of the biggest reasons people are skeptical about network automation is due to the lack of trust. Did this automation script/test really do what it's supposed to do? Triggers provide that verification out of the box through global and local verifications.

What if we wanted to build our own trigger with verifications? In the next section, you'll see how to do just that!

Writing a Trigger

The pyATS library (Genie) provides the ability to write your own triggers. A trigger is simply a Python class that has multiple tests in it that either configure, verify, or unconfigure the configuration or device feature you're trying to test.

To begin, your custom trigger must inherit from a base **Trigger** class. This base class contains common setup and cleanup tasks that help identify any unexpected changes to testbed devices not currently under testing (for example, a device rebooting). For our custom trigger, we are going to shut and unshut OSPF. Yes, this trigger already exists in the library, but it serves as a great example when you're beginning to create custom triggers. The workflow is going to look like this:

1. Check that OSPF is configured and running.
2. Shut down the OSPF process.
3. Verify that OSPF is shut down.
4. Unshut the OSPF process.
5. Verify OSPF is up and running.

In Examples 9-14 and 9-15, you'll see the code to create the custom OSPF trigger and the associated job file, running it with **gRun**. To run the job file, you'll need the following files:

- ex0915_custom_trigger.py
- ex0916_custom_trigger_job.py
- ex0915_custom_trigger_datafile.yaml
- testbed2.yaml

The testbed2.yaml file has two IOSv routers, named "iosv-0" and "iosv-1," running OSPF. The file ex0915_custom_trigger_datafile.yaml is used to map the custom OSPF triggers and the testbed devices:

```
# ex0915_custom_trigger_datafile.yaml

extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(trigger)}"

# Custom trigger
TriggerMyShutNoShutOspf:
    # source imports the custom trigger
    source:
        class: ex0915_custom_trigger.MyShutNoShutOspf
    devices: ["iosv-0", "iosv-1"]
```

Example 9-14 *Custom Trigger and Job File*

```
import time
import logging
from pyats import aetest

from genie.harness.base import Trigger
from genie.metaparser.util.exceptions import SchemaEmptyParserError

log = logging.getLogger()

class MyShutNoShutOspf(Trigger):
"""Shut and unshut OSPF process. Verify both actions."""

                @aetest.setup
                def prerequisites(self, uut):
                """Check whether OSPF is configured and running."""

                # Checks if OSPF is configured. If not, skip this trigger
                try:
                                output = uut.parse("show ip ospf")
                except SchemaEmptyParserError:
                    self.failed(f"OSPF is not configured on device {uut.name}")

                # Extract the OSPF process ID
                self.ospf_id = list(output["vrf"]["default"]["address_family"] \
                ["ipv4"]["instance"].keys())[0]
```

```
# Checks if the OSPF process is enabled
ospf_enabled = output["vrf"]["default"]["address_family"] \
["ipv4"]["instance"][self.ospf_id]["enable"]

if not ospf_enabled:
      self.skipped(f"OSPF is not enabled on device {uut.name}")


@aetest.test
def ShutOspf(self, uut):
"""Shutdown the OSPF process"""

uut.configure(f"router ospf {self.ospf_id}\n shutdown")
time.sleep(5)


@aetest.test
def verify_ShutOspf(self, uut):
"""Verify ShutOspf worked"""

output = uut.parse("show ip ospf")

ospf_enabled = output["vrf"]["default"]["address_family"] \
["ipv4"]["instance"][self.ospf_id]["enable"]

if ospf_enabled:
          self.failed(f"OSPF is enabled on device {uut.name}")

@aetest.test
def NoShutOspf(self, uut):
"""Unshut the OSPF process"""

uut.configure(f"router ospf {self.ospf_id}\n no shutdown")

@aetest.test
def verify_NoShutOspf(self, uut):
"""Verify NoShutOspf worked"""

output = uut.parse("show ip ospf")

ospf_enabled = output["vrf"]["default"]["address_family"] \
["ipv4"]["instance"][self.ospf_id]["enable"]

if not ospf_enabled:
           self.failed(f"OSPF is enabled on device {uut.name}")
```

Example 9-15 *Running a Custom Trigger – ex0915_custom_trigger_job.py*

```
from genie.harness.main import gRun


def main():
         gRun(
         trigger_uids=["TriggerMyShutNoShutOspf"],
         trigger_datafile="ex0915_custom_trigger_datafile.yaml",
         )




# Running the job using 'pyats run job' command
# pyats run job ex0916_custom_trigger_job.py --testbed-file testbed2.yaml
```

Figure 9-3 shows some sample job output.

```
+------------------------------------------------------------------------------+
|                            Task Result Details                               |
+------------------------------------------------------------------------------+
Task-1: genie_testscript
|-- common_setup                                                          PASSED
|   |-- connect                                                           PASSED
|   |-- configure                                                         SKIPPED
|   |-- configuration_snapshot                                            PASSED
|   |-- save_bootvar                                                      PASSED
|   |-- learn_system_defaults                                             PASSED
|   |-- initialize_traffic                                                SKIPPED
|   `-- PostProcessor-1                                                   PASSED
|-- TriggerMyShutNoShutOspf.iosv-0                                        PASSED
|   |-- prerequisites                                                     PASSED
|   |-- ShutOspf                                                          PASSED
|   |-- verify_ShutOspf                                                   PASSED
|   |-- NoShutOspf                                                        PASSED
|   `-- verify_NoShutOspf                                                 PASSED
|-- TriggerMyShutNoShutOspf.iosv-1                                        PASSED
|   |-- prerequisites                                                     PASSED
|   |-- ShutOspf                                                          PASSED
|   |-- verify_ShutOspf                                                   PASSED
|   |-- NoShutOspf                                                        PASSED
|   `-- verify_NoShutOspf                                                 PASSED
`-- common_cleanup                                                        PASSED
    |-- verify_configuration_snapshot                                     PASSED
    |-- stop_traffic                                                      SKIPPED
    `-- PostProcessor-1                                                   PASSED
```

Figure 9-3 *Job Results*

Trigger and Verification Example

Now it's time to combine the triggers and verifications into one complete example. In the example, we will build on previous examples where we flap (shut/unshut) the OSPF process on two IOSv routers. The trigger has local verifications that will confirm OSPF

is indeed shut down and confirm that it comes up after being unshut. In addition to the local verifications defined in the trigger, the job will also introduce a global verification to check the router link states (LSA Type 1) in the OSPF LSDB (link state database). This global verification will run before and after the trigger. This allows us to confirm that LSA Type 1 packets are being exchanged before and after testing and that the router link types have not changed during testing.

Let's see how this example can be implemented and executed using Genie Harness and also within a pyATS testscript.

Genie Harness (gRun)

This whole chapter has been focused on Genie Harness, so let's not rehash the details. Example 9-16 shows an example of the job file, trigger datafile, and verification datafile used to execute the job. Figure 9-4 shows the associated test results.

Example 9-16 *Trigger and Verification Example – ex0917_complete_example.py*

```
# Job file - ex0917_complete_example.py
from genie.harness.main import gRun

def main():
gRun(
         trigger_uids=["TriggerShutNoShutOspf"],
         trigger_datafile="ex0917_trigger_datafile.yaml",
         verification_uids=["Verify_IpOspfDatabaseRouter"],
         verification_datafile="ex0917_verification_datafile.yaml"
         )



# Trigger datafile – ex0917_trigger_datafile.yaml
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(trigger)}"

# Custom trigger - created before Example 9-14
TriggerShutNoShutOspf:
# source imports the custom trigger, just as you would any other Python class
  source:
    class: ex0915_custom_trigger.MyShutNoShutOspf
  devices: ["iosv-0", "iosv-1"]
```

```
# Verification datafile - ex0917_verification_datafile.yaml
extends: "%CALLABLE{genie.libs.sdk.genie_yamls.datafile(verification)}"


Verify_IpOspfDatabaseRouter:
  devices: ["iosv-0", "iosv-1"]
```

```
+------------------------------------------------------------------------------+
|                            Task Result Summary                               |
+------------------------------------------------------------------------------+
Task-1: genie_testscript.common_setup                                    PASSED
Task-1: genie_testscript.Verifications.TriggerShutNoShutOspf.iosv-0      PASSED
Task-1: genie_testscript.TriggerShutNoShutOspf.iosv-0                    PASSED
Task-1: genie_testscript.Verifications.TriggerShutNoShutOspf.iosv-1      PASSED
Task-1: genie_testscript.TriggerShutNoShutOspf.iosv-1                    PASSED
Task-1: genie_testscript.Verifications.post                              PASSED
Task-1: genie_testscript.common_cleanup                                  PASSED

+------------------------------------------------------------------------------+
|                            Task Result Details                               |
+------------------------------------------------------------------------------+
Task-1: genie_testscript
|-- common_setup                                                         PASSED
|   |-- connect                                                          PASSED
|   |-- configure                                                        SKIPPED
|   |-- configuration_snapshot                                           PASSED
|   |-- save_bootvar                                                     PASSED
|   |-- learn_system_defaults                                            PASSED
|   |-- initialize_traffic                                               SKIPPED
|   `-- PostProcessor-1                                                  PASSED
|-- Verifications.TriggerShutNoShutOspf.iosv-0                           PASSED
|   |-- Verify_IpOspfDatabaseRouter.iosv-0.1                             PASSED
|   |   `-- verify                                                       PASSED
|   `-- Verify_IpOspfDatabaseRouter.iosv-1.1                             PASSED
|       `-- verify                                                       PASSED
|-- TriggerShutNoShutOspf.iosv-0                                         PASSED
|   |-- prerequisites                                                    PASSED
|   |-- ShutOspf                                                         PASSED
|   |-- verify_ShutOspf                                                  PASSED
|   |-- NoShutOspf                                                       PASSED
|   `-- verify_NoShutOspf                                                PASSED
|-- Verifications.TriggerShutNoShutOspf.iosv-1                           PASSED
|   |-- Verify_IpOspfDatabaseRouter.iosv-0.2                             PASSED
|   |   `-- verify                                                       PASSED
|   `-- Verify_IpOspfDatabaseRouter.iosv-1.2                             PASSED
|       `-- verify                                                       PASSED
|-- TriggerShutNoShutOspf.iosv-1                                         PASSED
|   |-- prerequisites                                                    PASSED
|   |-- ShutOspf                                                         PASSED
|   |-- verify_ShutOspf                                                  PASSED
|   |-- NoShutOspf                                                       PASSED
|   `-- verify_NoShutOspf                                                PASSED
|-- Verifications.post                                                   PASSED
|   |-- Verify_IpOspfDatabaseRouter.iosv-0.3                             PASSED
|   |   `-- verify                                                       PASSED
|   `-- Verify_IpOspfDatabaseRouter.iosv-1.3                             PASSED
|       `-- verify                                                       PASSED
`-- common_cleanup                                                       PASSED
    |-- verify_configuration_snapshot                                    PASSED
    |-- stop_traffic                                                     SKIPPED
    `-- PostProcessor-1                                                  PASSED
```

Figure 9-4 *Trigger and Verification Example Results*

pyATS

Triggers and verifications can be run in a pyATS testscript as a testcase or within a test section. Custom trigger and verification datafiles can be provided using the **--trigger-datafile** and **--verification-datafile** arguments when calling a pyATS job.

To run it as its own testcase, you'll just need to create a class that inherits from **GenieStandalone**, which inherits from the pyATS **Testcase** class. The inherited class you create will provide a list of triggers and verifications. The same trigger and verification datafiles will be included as options when running the pyATS job via the command line.

The second way to include triggers and verifications in a pyATS testscript is by including them in an individual test section, as part of a pyATS testcase. The **run_genie_sdk** function allows you to run triggers or verifications as steps within a section.

Example 9-17 shows how to include triggers and verifications as their own testcase and within an existing subsection in a pyATS testscript. To run the testscript, you'll need to add the **--trigger-datafile** and **--verification-datafile** arguments with the appropriate datafiles to map the custom trigger and verifications to the additional devices. These datafiles are not included in the example. Figure 9-5 shows the testscript results.

Example 9-17 *Triggers and Verifications in pyATS Testscript*

```
from pyats import aetest
import genie
from genie.harness.standalone import GenieStandalone, run_genie_sdk


class CommonSetup(aetest.CommonSetup):
""" Common Setup section """

       @aetest.subsection
       def connect(self, testbed):
       """Connect to each device in the testbed."""

               genie_testbed = genie.testbed.load(testbed)
               self.parent.parameters["testbed"] = genie_testbed
               genie_testbed.connect()


# Call Triggers and Verifications as independent pyATS testcase
class GenieOspfTriggerVerification(GenieStandalone):
"""Shut/unshut the OSPF process and verify LSA Type 1 packets are still being
exchanged before and after testing."""
```

```
        # Must specify 'uut'
        # If other devices are included in the datafile(s), they will be tested
        uut = "iosv-0"
        triggers = ["TriggerShutNoShutOspf"]
        verifications = ["Verify_IpOspfDatabaseRouter"]




# Calling Triggers and Verifications within a pyATS section
class tc_pyats_genie(aetest.Testcase):
        """Testcase with triggers and verifications."""
        # First test section
        @aetest.test
        def simple_test_1(self, steps):
        """Sample test section."""

        # Run Genie triggers and verifications
        # Note that you must specify the order of each trigger and verification
        run_genie_sdk(self,
                        steps,
                        ["Verify_IpOspfDatabaseRouter", \
                         "TriggerShutNoShutOspf", \
                         "Verify_IpOspfDatabaseRouter"],
                        uut="iosv-0"
                        )


class CommonCleanup(aetest.CommonCleanup):
"""Common Cleanup section"""

@aetest.subsection
def disconnect_from_devices(self, testbed):
        """Disconnect from each device in the testbed."""
        testbed.disconnect()
```

```
+------------------------------------------------------------------------------+
|                            Task Result Summary                               |
+------------------------------------------------------------------------------+
Task-1: ex0918_pyats_testscript.common_setup                              PASSED
Task-1: ex0918_pyats_testscript.GenieOspfTriggerVerification              PASSED
Task-1: ex0918_pyats_testscript.tc_pyats_genie                            PASSED
Task-1: ex0918_pyats_testscript.common_cleanup                            PASSED

+------------------------------------------------------------------------------+
|                            Task Result Details                               |
+------------------------------------------------------------------------------+
Task-1: ex0918_pyats_testscript
|-- common_setup                                                          PASSED
|   `-- connect                                                           PASSED
|-- GenieOspfTriggerVerification                                          PASSED
|   |-- Verify_IpOspfDatabaseRouter.iosv-0_verify.1                       PASSED
|   |-- Verify_IpOspfDatabaseRouter.iosv-1_verify.2                       PASSED
|   |-- TriggerShutNoShutOspf.iosv-0_prerequisites.3                      PASSED
|   |-- TriggerShutNoShutOspf.iosv-0_ShutOspf.4                           PASSED
|   |-- TriggerShutNoShutOspf.iosv-0_verify_ShutOspf.5                    PASSED
|   |-- TriggerShutNoShutOspf.iosv-0_NoShutOspf.6                         PASSED
|   |-- TriggerShutNoShutOspf.iosv-0_verify_NoShutOspf.7                  PASSED
|   |-- TriggerShutNoShutOspf.iosv-1_prerequisites.8                      PASSED
|   |-- TriggerShutNoShutOspf.iosv-1_ShutOspf.9                           PASSED
|   |-- TriggerShutNoShutOspf.iosv-1_verify_ShutOspf.10                   PASSED
|   |-- TriggerShutNoShutOspf.iosv-1_NoShutOspf.11                        PASSED
|   |-- TriggerShutNoShutOspf.iosv-1_verify_NoShutOspf.12                 PASSED
|   |-- Verify_IpOspfDatabaseRouter.iosv-0_verify.13                      PASSED
|   `-- Verify_IpOspfDatabaseRouter.iosv-1_verify.14                      PASSED
|-- tc_pyats_genie                                                        PASSED
|   `-- simple_test_1                                                     PASSED
|       |-- STEP 1: Verify_IpOspfDatabaseRouter.iosv-0                    PASSED
|       |-- STEP 2: TriggerShutNoShutOspf.iosv-0                          PASSED
|       `-- STEP 3: Verify_IpOspfDatabaseRouter.iosv-0                    PASSED
`-- common_cleanup                                                        PASSED
    `-- disconnect_from_devices                                           PASSED
```

Figure 9-5 *Triggers and Verifications in pyATS Testscript Results*

Summary

This chapter covered a lot of information about the pyATS library (Genie), the Genie Harness, with its many different features, along with triggers and verifications. The Genie Harness allows you to take advantage of the pyATS infrastructure without having to dive-deep into code. The goal of the pyATS library (Genie) is to be modular and robust. Triggers and verifications are a perfect example. They make it easy to quickly build dynamic testcases that change with your network requirements. I highly recommend taking a closer at the code examples in this chapter and trying them out for yourself. Within minutes, you'll see how quickly you can test a network feature with speed and accuracy!

This page intentionally left blank

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q-R

S

T

U

V

W

X

Y

Z